IMPROVING THE THROUGHPUT OF NOVEL CLUSTER COMPUTING SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Jiadong Wu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2015

IMPROVING THE THROUGHPUT OF NOVEL CLUSTER COMPUTING SYSTEMS

Approved by:

Professor Bo Hong, Committee Chair
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Bo Hong, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Douglas Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Linda Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Saibal Mukhopadhyay
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Yi-Chang James Tsai
School of Civil Engineering
Georgia Institute of Technology

Date Approved: 14 May 2014

To my wife,

Fan Yun,

For all of her love and support.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my greatest appreciation to people who have helped me. Without their support, encouragement, and love, my doctoral study would not have been such a joyful journey.

My thank first goes to my advisor Dr. Hong and my fellow colleagues Zhengyu He, Xiao Yu, and Weiming Shi. They introduced me to the richness and depth of parallel computing. Their sparkling ideas have always inspired me. I am very grateful to Dr. Hong for his guidance, for sharing his wisdom and breadth of knowledge, and for his patience in allowing me to explore. I am also grateful to my colleagues for their great collaboration and assistance during my study.

I would like to thank my committee Dr. Blough, Dr. Wills, Dr. Mukhopadhyay, and Dr. Tsai for taking time and effort to give me insightful suggestions and strong guidance on my dissertation.

Last but not least, I owe my Ph.D. degree to my dearest wife Fan Yun. She believes in me, cherishes me, and lightens up my life. Without her support and sacrifice, this work would not have been possible. No word can express my deepest gratitude for what she has done for me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

As a natural extension to the stand-alone computer, the computer cluster has attracted enormous interest over half a century [101]. Motivated by its obvious advantages in computation speed, storage capacity, high-availability, and load-balance, the computer cluster technology has been widely applied in high-performance computing [23], database management [36], Internet service [19], and cloud infrastructure [24].

The cluster computing systems are a type of clusters that are developed primarily for computation. One of the best examples of such system is the modern supercomputer, in which a large set of computer nodes are connected by high-performance network to solve complex computational problems. Currently, over 85% of the world's fastest supercomputers are cluster computing systems [120]. From bio-informatics study [116] to astrophysics simulation [55], from health care [71] to business intelligence [90], a broad range of scientific and social endeavors are depending on the advancement of cluster computing systems.

Traditional computing clusters such as the supercomputers are powered by specially designed high-performance hardware, which escalates the manufacturing cost and the energy consumption of those systems. The past decade has seen the emergence of several prominent cluster computing systems that are driven by specialized application demands and cost-effective hardware/software technologies. In this research, we are focusing on the high-performance GPU cluster [42] and the commodity Hadoop cluster [54]. The GPU cluster is developed to accelerate computation-intensive applications by combining traditional computing cluster with high-performance

GPU. Thanks to the massively-parallel architecture of GPU, this system can deliver much higher performance-per-watt than the traditional CPU-only systems. The Hadoop cluster is developed with off-the-shelf hardware component and standard network to cost-effectively handle data-intensive applications. A series of brilliant software techniques, such as locality-aware job scheduling and automatic data replication, are applied in the system to optimize performance and availability.

1.1 Research Objective

The objective of this research is to improve the throughput of cluster computing systems by enhancing the utilization of key resources – the processors and the network. According to our study, even though the novel computing clusters are specialized and cost-effective, their performance can be seriously compromised by the inefficiencies involved in resource utilization.

In the GPU cluster, even thought both CPU and GPU are capable of doing computation, these two processing units are essentially designed for different workloads. To achieve a reasonable performance, the application developers have to carefully decide which part of their job is executed on CPU and which part is on GPU based on the characteristics of the workload. On the other hand, however, the hardware CPU-to-GPU ratio of any cluster is fixed. The intrinsic mismatch between the workload-dependent demand and the platform-dependent availability can waste a significant portion of the system's computation capacity. In this research, we aim to lower such mismatch by enabling flexible resource sharing and increasing the job parallelism.

In the commodity Hadoop cluster, the low-performance network is a major bottleneck. Although the Hadoop system is designed to reduce network traffic by moving computation to data, the contention over network resources is inevitable. In this research, we are focusing on the limitations of existing Hadoop system in handling remote data I/O between the application and the underlying distributed file system.

Due to the replication-based design of Hadoop, we are trying to explore the opportunities of leveraging the parallelism inside the source and destination of data traffic to alleviate network congestion and improve throughput of the entire cluster.

1.2 Summary of Contributions

The main contributions of this research can be summarized as follows.

1. We profiled the job traces of real medium-scale GPU cluster and identified the serious underutilization problem caused by the heterogeneity among computation resources. Our model analysis of existing GPU clusters revealed that the limitations in resource invocation and job scheduling policy were preventing the CPUs and GPUs from being efficiently utilized [138, 134].

2. To enhance the GPU utilization, we lifted the restriction of static mapping kernel-device in existing GPU clusters. We created a GPU resource management framework that combines the remote GPU kernel execution technologies with dynamic kernel-device mapping policies to balance the GPU utilization across the cluster. We studied the potential overheads involved in different execution conditions resulted from dynamic mapping, and designed two adaptive polices to maximize the throughput. The performance of the policies were evaluated with our GPU cluster simulator and synthesized workload. The results showed that the adaptive policies can efficiently shift GPU workload from congested GPU devices to idling devices over the network and improve the cluster-wide GPU utilization by 5-15% [138].

3. To enhance the CPU utilization, we systematically evaluated the possibilities of running GPU-assisted job and CPU-only job together on over-subscribe CPU cores, and pinpointed a set of factors that would cause major performance degradation for the jobs. Based on the evaluation, we designed a novel job

scheduling system for GPU clusters to smartly co-locate heterogeneous jobs. The experiments on our 4-node cluster proved that the new system can effectively detect and scavenge the underutilized CPU resources. With a moderate resource collection setting, the system achieved 15.9% gain in throughput and 10% gain in both CPU and GPU utilizations [134].

4. We studied the limitations of the static single-source remote data accessing mechanism in existing Hadoop clusters, and summarized the cases in which such mechanism could compromise the performance of specific Hadoop application as well as the entire system. To accelerate the remote access, we created a dynamic multi-source streaming technique that could leverage the availability of data replicas in a Hadoop cluster. This novel approach first slices the data and then streams the pieces simultaneously from multiple sources. We profiled the network performance of several different cluster configurations (including bare metal cluster, virtualized cluster, and cloud-based cluster), and designed the adaptive heuristics of slicing and streaming. Our experiments showed that multi-source streaming could help the applications boost remote access throughput by 10-20% on various clusters. Moreover, because of its unique ability to adaptively explore data locality, the multi-source streaming technique demonstrated great potential in accelerating the cloud-based Hadoop clusters [135, 136].

5. We highlighted the redundant traffic incurred by the pipelined replication mechanism in existing Hadoop clusters, and discussed the opportunities of using multicast technology to reduce such traffic and alleviate the network congestion. We created a congestion-controlled reliable multicast socket in Java, and implemented Hadoop's data replication functionality on top of it. The new system was tested on our 24-node multi-rack cluster, and the results indicated

that the multicast-based replication mechanism can reduce network traffic and can coexist with TCP. To further evaluate the impact of multicast to large-scale systems, we designed a flow-based Hadoop cluster simulator that was running on real workload trace from Facebook's production Hadoop cluster. The simulation result confirmed that the multicast-base replication can increase the throughput of such large Hadoop cluster by as much as 12% [137].

1.3 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 provides background information on cluster computing system as well as other types of computer clusters. Chapter 3 gives an overview of GPU cluster and studies its underutilization problem. The dynamic GPU kernel-device mapping technique is then presented in Chapter 4 to improve GPU utilization, and the heterogeneous job collocation technique is presented in Chapter 5 to improve CPU utilization. Chapter 6 gives an overview of Hadoop cluster and also discusses the limitations of existing Hadoop system. The enhancements of multi-source streaming and multicast-based replication are presented in Chapter 7 and Chapter 8 respectively.

CHAPTER II

BACKGROUND OF COMPUTER CLUSTERS

A computer cluster is a collection of computers closely connected by a fast local area network (LAN) and consistently orchestrated to work as a single system. Usually, all the nodes of a computer cluster are configured with identical hardware and software to maximize interoperability and minimize performance variation. Unlike other multi-node parallel and distributed systems, such as the peer-to-peer systems and the grid computing systems, the nodes in a computer cluster are usually located in a single geological site and managed by centralized software appliances.

The origin of computer clusters can be traced back to early 1960s [101], since the idea of clustering is so intuitive – using multiple connected computers to solve a problem that is too large for single computer. The VAXcluster [76] launched DEC by in 1983 was one of the first commercially available computer clusters. In that cluster, a proprietary network switch was used to connect all the computers and storage devices. Each computer had its own memory and ran its own VMS operating system. A distributed lock manager was developed to coordinate the accesses to the shared storage. The VAXcluster was able to operate as a batch processing system and survive node failure. Another milestone in the history of computer cluster is the Beowulf system [21], which was developed by Thomas Sterling and Donald Becker in 1994. By relying on 100% commodity hardware, standard network, and open source software, this system marked the birth of inexpensive vendor-independent high-performance cluster computing systems.

Although the computer clusters can be designed with different sizes, architectures, and technologies, they are all driven by the same set of motivations as listed below.

1. *Computation Speed – the computers are connected to solve a complex problem.* According to the Amdahl's law, a computer cluster can drastically speedup the parallelizable applications by distributing the workload to a large number of nodes.

2. *Storage Capacity – the computers are connected to store a large volume of data.* In a computer cluster, the individually attached storage devices (e.g. RAM, flash memory, and disk) can be united to provide a single storage image. In addition to the extended storage capacity, such union also enables the cluster to provide high parallel throughput.

3. *High Availability – the computers are connected to avoid single point of failure.* In a computer system, any piece of hardware would fail indefinitely. However, with properly designed fault-tolerance software, a computer cluster can handle the failures gracefully and provide the applications with exceptional availability.

4. *Load Balance – the computers are connected to balance the load.* Connecting the computers into a cluster can help even out the workload among individual computers and thus increase the aggregated performance of the system. It also enables the resources to be elastically added/removed from the system.

These motivations are orthogonal, so that a real computer cluster can incorporate any combination of the four to address the demand of specific applications.

2.1 Cluster Computing Systems

The focus of this research is the cluster computing systems, which are a specific type of computer clusters developed primarily for computational purpose. As exhibited in Figure 1, the framework of a cluster computer system includes following major components.

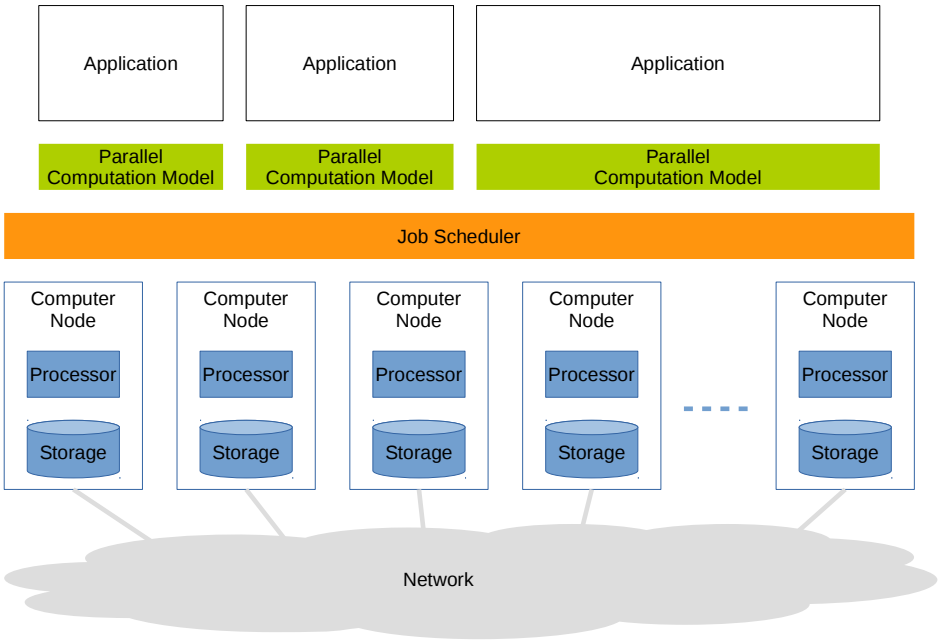

Figure 1: Framework of Cluster Computing System

2.1.1 Hardware Stack

2.1.1.1 Processor

The computer nodes in almost all the modern cluster computing systems are powered by multi-core CPUs. According to the latest Top500 [120] list of supercomputers, more than 85% of the systems are based on commodity multi-core CPUs including the Intel Xeon family and the AMD Opteron family. A small fraction of the systems are based on customized multi-core processors including the 8-core SPARC64 CPU in Fujitsu K computer cluster. Each of these CPUs typically has 4 to 16 identical processor cores running at 2 3GHz frequency. A set of fast caches are attached to the cores and multiple high-level shared caches are also available. The multi-core CPUs are designed as MIMD (Multiple-Instruction-Multiple-Data) devices that are extremely effective in running sequential programs (at high-frequency) and handling irregular memory accesses (with multi-level cache).

In addition to the CPU, accelerator technologies are also gaining popularity.

Among which, GPU is the most widely used accelerator in today's cluster computing systems [42]. Around 10% of the Top500 systems are already equipped with GPU. Unlike the CPU, which is optimized for sequential workload, the GPU is designed as a SIMD (Single-Instruction-Multiple-Data) device and optimized for massively-parallel workload. Taking the Nvidia K40 GPU as an example, a modern GPU can have as many as 2880 cores running at about 800MHz. The cores are evenly divided into 15 groups (dubbed as SM or Streaming Multiprocessor). Inside each SM, there are 64KB fast shared memory that can be accessed deliberately by the 192 cores or automatically as a L1 cache. A global memory of 12GB is available on the K40 GPU device. Such many-core architecture enables GPU to achieve much higher parallel performance than CPU while consuming much lower energy. According to the Green500 list [44] that ranks the supercomputers by energy efficiency, GPU clusters are among the most efficient systems for computation-intensive applications.

On the other hand, however, the introduction of GPU inevitably creates heterogeneity in computation resources of a cluster computing system, because the execution model of GPU is completely different from that of CPU. GPU achieves massive parallelism through simultaneously launching of a large number of lightweight threads. The GPU threads are scheduled onto processor cores in the unit of a warp (32 threads) rather than of a single thread [94]. Different warps execute independently regardless of whether they are executing common or disjoint code paths. Threads within a warp execute the same instruction at a time but on different pieces of data, so maximum efficiency is achieved when all threads of a warp have identical execution path. If a warp of threads diverge on data-dependent conditional branches, the warp will sequentially execute all the branch paths, and subsequently nullify results from threads that are not on that path. The threads converge back to the same execution path after all paths are completed. Note that diverged execution paths often cause significant degradation to the efficiency of GPU programs.

In summary, only if the workload fits the SIMD model (with few diverged execution paths) can GPU achieve substantial performance advantage. For sequential and MIMD workload, CPU is definitely faster. Moreover, the isolation of CPU memory and GPU memory further increases the overhead for the workload to switch between the two processors. Such heterogeneity can incur serious utilization problem for the cluster computing systems that are equipped with GPU as well as other accelerator alternatives including Intel Xeon Phi [68] and FPGA [25].

2.1.1.2 Network

Network is another critical component of any cluster computing system. Different applications tend to have different demand over network bandwidth and latency, so network infrastructure varies a lot among the clusters.

The basic form of cluster network is Gigabit Ethernet (GbE). As the standard network interface for modern PC and workstation, GbE provides relatively high bandwidth with the most affordable price. A large number of computers (100s-1000s) can be connected into a hierarchical tree, with GbE switches as the tree nodes and the computers as the leaves. The problem of GbE involves the long latency and the limited bandwidth between the leaves of different tree nodes. The growing availability of 10 40Gbps layer 2/3 switches partially alleviates the problem by enabling the Ethernet-based clusters to adopt the high-bandwidth fat-tree topology [6]. In the Top500 supercomputers, about 38% of the system are based on Ethernet. In enterprise data centers, Ethernet is expected to be the dominant form of network.

The high-performance computing clusters typically relies on low-latency interconnections. The standardized option is the Infiniband [13], which has been used by 45% of the Top500 supercomputers. The Infiniband network is based on a fat-tree topology similar to the Ethernet but can offer lower latency. Several proprietary network

options are available for high-end systems, which include the 6D-torus interconnection [5] in Fujitsu K computer cluster, and the 2-tier dragonfly interconnection [73] in Cray XC30. However, the extensive manufacturing cost of high-bandwidth low-latency network equipments directly prevents these technologies from being widely adopted.

2.1.1.3 Storage

There are two major storage options for cluster computing systems: dedicated storage system and consolidated storage system.

In high-performance computing clusters, a subset of nodes are usually used as dedicated storage nodes, on which a distributed file system is created to host the input/output data of entire cluster. The storage nodes are sometimes connected by a private sub-network to further reduce the interference of I/O traffic to the main network. Lustre [22] is the most popular distributed file system for high-performance computing system. It has been used by more than 75% of the top 100 supercomputers. A typical Lustre system consists of a pair of metadata servers and multiple object storage servers. The metadata server maps a specific file to a fixed set of object storage servers. When the file is being accessed, the actual data is written to and read from the object storage servers directly. Both the metadata servers and the object storage servers use redundant array of independent disks (RAID) or storage area network (SAN) as their repositories to guarantee the data reliability. However, due to the utilization of multiple high-performance hardware (e.g. private network, RAID, and SAN), the cost of deploying a Lustre-like dedicated storage system is quite high.

With the emergence of big-data applications [86], the demand for a cost-effective storage for cluster computing systems is greater than ever before. Extraordinary efforts have been made to consolidate the storage functionalities onto the computing

nodes. This approach (sometimes known as shared-nothing architecture) can drastically increase the number of equivalent storage nodes and reduce the overall hardware cost. Moreover, consolidating storage and computation also enables the applications to exploit data locality, such that the data can be accessed locally without incurring network overhead. The Hadoop Distributed File System [113] (HDFS) is a widely used consolidated storage system for computing clusters. The design of HDFS is inspired by the Google File System [48], in which the actual data is formatted as objects and stored on the computer nodes across the cluster. A centralized sever manages all the metadata and provides a file system interface the applications. HDFS is different from Lustre that (1) the data reliability is achieved by software techniques (i.e. replication and rack-aware data placement) rather than expensive hardware, and (2) the physical location of the data is exposed to the applications.

2.1.2 Software Stack

2.1.2.1 Parallel Programming Model

The Message Passing Interface [52] (MPI) is the most widely used programming model for the cluster computing systems. A MPI job is a collection of processes that are started at the beginning of the job. The model assumes that each process has its private processor core and memory space so that different programs can be executed on the processes. The MPI processes coordinate their progress by exchanging messages via the local memory and/or the network. The messages can be delivered point-to-point or in a collective manner. This model can effectively exploit the inherent parallelism of a cluster computing system, because it precisely resembles the structure of such system and the MIMD processors inside of it.

The Compute Unified Device Architecture [94] (CUDA) is a programming model dedicated to the general-purpose GPU computing. It exposes GPU's four major architectural features to the developers: the hierarchy of processor groups, the barrier synchronization functionality, the shared memories, and the global memory. A basic

CUDA application consists of a host process and a GPU computation kernel. The host process is running on CPU and is responsible for moving data into and out of the GPU's global memory. The GPU kernel, which appears to the host process as a function call, is a SIMD program for the GPU device. Due to the limitation of CUDA, a kernel is statically bound to one specific GPU device on local computer node. The application workload is described in the kernel as a large set (1000s or more) of identical threads that are working on different pieces of data according to their distinct thread IDs. The threads inside the same hierarchical group can communicate via the shared memory and synchronize using a barrier. The threads of different groups can be coordinated (with substantial overhead) through atomic operations over the global memory. The combination of MPI and CUDA provides a straightforward offloading model for programming the GPU-equipped cluster computing system. In such model, a MPI process will stick all its sequential workload to the CPU and offload any data parallel workload (such as a big matrix multiplication) to the GPU.

MapReduce is a specialized SIMD-like programming model [34]designed for large-scale data-intensive workload. It relies on only two tasks: map and reduce. A MapReduce job incorporates a set of map tasks and reduce tasks, which are written by the application developers. Both the input and output of the tasks are key/value pairs. Each map task will take a logical split of the input and generate a collection of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with each specific key and passes them to the reduce function. Each reduce task will then take a subset of all the intermediate keys (with their associated values) and generate new key/values pairs as output. In the MapReduce model, the only means of synchronization is the implicit barrier between the map phase and the reduce phase, and the only means of communication is the intermediate data shuffled during the transition. Based on this model, programmers can write data-intensive jobs in concise language, and let the underlying MapReduce library, such

as the Hadoop MapReduce [54], schedule and execute the jobs on the large clusters reliably.

2.1.2.2 Job Scheduler

Although there are a few clusters that are dedicated to single application, the majority of the computing clusters are developed as batch processing systems to serve the computation requests of a broad user base. In general, such system usually consists of a centralized manager, who maintains a pool of resources and a queue of jobs, and a set of distributed monitors, who oversee the individual computers and the job execution. The core of a batch processing system is the scheduling policy over jobs and resources.

The Portable Batch System [59] (PBS) is a popular batch processing system for computing clusters. PBS supports multiple prioritized job queues and pluggable scheduling policies. When submitting a job to PBS, the user would usually specify the amount of resource needed (e.g. processor count, memory size), the estimated execution time, and the associated job queue. When any piece of resource is freed, the system would scan the queues and see if a new job can be scheduled according to specific policy [67]. The minimal condition for a job to be scheduled is the availability of all the resources it requested. Since PBS is originally developed for CPU-only clusters, it is usually configured to maximize CPU utilization without oversubscribing any resource.

The Hadoop system (based on the MapReduce programming model and the consolidated storage HDFS) is another widely used batch processing system for computing clusters. Due to the specialized programming model and storage architecture, the scheduling of Hadoop is different from PBS in several ways: (1) the objective is to balance resource utilization with data locality; (2) the job comes with no specification about resources or execution time; (3) the job can start execution with a tiny

slot of resources; (4) The resources are usually shared and over-subscribed. However, Hadoop and PBS share the fundamental scheduling goal – to optimize the overall throughput of the computing cluster.

2.1.2.3 Applications

The cluster computing applications can be roughly broken into two categories: computation-intensive and data-intensive. For much of the history, the high-performance cluster computing systems are developed to solve complicated scientific problems including bioinformatics [116], astrophysics [55], and seismology [100]. Because of their high computation complexity, these applications usually require a massive amount of CPU, memory, and network resources. Such a challenging demand is the ultimate driving force behind many cutting-edge cluster technologies, including the rapid development of high-performance GPU clusters.

On the other hand, due to the recent emergence of data-intensive applications, more and more computing clusters are being used for these low-complexity high-volume problems. Taking the word count application as an example, which counts the appearance of each individual word in terabytes of text files, the solution is rather straightforward – split the files across multiple computers, count the appearance from each split, and then merge the results. The execution time of word count depends entirely on how fast the files are accessed. As the Internet services getting popular, a large quantity of similar big-data analytic problems are piled up in front of companies such as Google, Yahoo, and Facebook. Although the traditional high-performance computing clusters are more than capable of dealing with these problems, they are not necessarily the most cost-effective option. To address this demand, a series of data-oriented cluster technologies [20, 121, 78, 75] are developed by the Internet companies and the open source community. Among them, the commodity Hadoop system is currently the most widely adopted cluster computing system for

data-intensive applications.

2.2 Research Scope

In this research we are focusing on improving the throughput of computing cluster through better utilizing the heterogeneous computation resources and the congested network resources. The research is conducted in the context of two specific cluster computing systems – the high-performance GPU cluster and the commodity Hadoop cluster – due to their substantial exposure to the problems, extensive coverage over applications, and strong popularity.

1. *Exposure* The performance of these two systems are directly related to the problems that we focused on. In general, any cluster computing system can suffer from a under-performed network. However, the data-intensive orientation and the Ethernet-based architecture make the commodity Hadoop clusters especially vulnerable to network congestion. The GPU clusters, on the other hand, exhibit higher heterogeneity than other accelerator-based system.

2. *Coverage* The GPU clusters and the Hadoop clusters are designed to handle computation-intensive workload and data-intensive workload respectively. Together, these two systems can cover a majority of all the applications faced by today's cluster computing systems.

3. *Popularity* Due to their exceptional performance and cost-effective design, the GPU clusters and the Hadoop clusters are widely used. According to the Top500 list and the Green500 list, GPU cluster currently accounts for more than 10% of world's fastest supercomputers and 18 out of 20 most energy-efficient supercomputers. According to the Apache Hadoop website [54], Yahoo alone has more than 40,000 nodes running Hadoop. A survey [91] by the IDC group also pointed out that more than 65% of the responded enterprises either have

deployed or are willing to deploy Hadoop cluster in the near future.

2.3 Other Types of Cluster Systems

Besides the computing cluster, there are several other types of computer clusters. In this section, we briefly introduce the clusters developed for storage, database, and virtualization purposes. We also discuss how these clusters can benefit from our research.

2.3.1 Storage Cluster

Instead of being part of a computing cluster, a storage cluster can also operate as a stand-alone system hosting data for non-computational applications such as file sharing, data backup, and web services.

xFS [128], one of the earliest cluster storage system, provides file system interface over a flat server-less group of computers. The file management responsibility is distributed across multiple nodes to achieve good load balance and high availability. To coordinate the accesses from multiple users, xFS uses a log-structured data storage and supports a finer-grain locking. Petal [79] is another early system for storage clusters. It creates a unified virtual disk interface over multiple distributed block devices (i.e. disks). Each piece of data is duplicated on two disks for availability. Multiple users can share the storage by using the Frangipani file system implemented on top of Petal. Consistency is reached with a distributed locking service that serializes the accesses from multiple Frangipani servers to the virtual disk of Petal.

Besides Lustre and Hadoop HDFS, there are several other prominent modern cluster storage systems. Dynamo [35] is Amazon's highly available object-based storage system with a flat server-less design. Dynamo uses consistent hashing to place the data objects. The objects can be replicated and placed on the consistent hashing ring. Dynamo allows read and write operations to continue even during network partitions and resolves conflicts using multiple resolution mechanisms. Ceph [131] is designed

as a multi-interface storage system. Unlike other systems that support only one of the storage interface (i.e. block, file system, or object-based), Ceph provides all three interfaces on top of its distributed object-based storage backbone. Ramcloud [96] is an in-memory cluster storage system optimized for small data objects. It stores all the data in the computer nodes' main memory to provide ultra-low latency. It also maintains backup copies of data on secondary storage to ensure a high level of durability and availability.

Since many storage clusters are using similar data replication technique as HDFS, our multi-source streaming and multicast-based replication mechanisms are expected to be applicable to these systems.

2.3.2 Database Cluster

Another important type of computer cluster is the database cluster, which is essentially a combination of distributed query engines and object-based storage cluster. Although it shares many design features with the storage cluster and the computing cluster, the database cluster usually provides stronger transactional support (i.e. ACID – Atomicity, Consistency, Isolation, Durability) than normal storage clusters, and the database cluster can only process data queries rather than general computation jobs.

A straightforward form of database cluster, such as the Oracle RAC system [114], comes with multiple query servers and a shared storage (or storage cluster). To alleviate the potential heavy contention over shared storage, the RAC system builds a shared cache on each of the computing nodes and uses high-performance network to maintain cache coherence.

The majority of database clusters, such as the widely used MySQL cluster [109], are based on the shared-nothing architecture, in which the data is partitioned and each node will operate exclusively on a part of it. H-Store [70] is a cluster database

system designed for online transaction processing. Similar to the Ramcloud storage system, it stores the data in the main memories to improve the access latency and replicates the data in disks to ensure data durability. The in-memory key-value storage Redis [143] and the document storage MongoDB [29] are two very popular forms of noSQL database clusters. Due to their relaxed transactional design, these systems can scale out to far more computer nodes than MySQL and H-Store.

For the database clusters, our proposal of multi-source streaming and multicast-based replication is unlikely to offer much benefit, since latency is usually more important than throughput. According to several recent studies [60, 140, 133], GPU has exhibited some unique capabilities in accelerating database quires. It is possible in the near future that the database clusters may also suffer from the problem of resource heterogeneity.

2.3.3 Virtualization Cluster

The virtualization clusters are a relatively new type of computer cluster. Rather than providing computation or data as services, the virtualiztion clusters are developed to provide virtual machines as service. The creation of virtualization cluster is driven by the trend of resource consolidation in enterprise data centers (e.g. Vmware vCenter [33]) and cloud computing (e.g. Amazon EC2 [8]). In these clusters, every computer node has its own hypervisor, which directly controls the virtual machines running on top of it. All the hypervisors are then controlled by a centralized provisioning system, so that the load can be effectively balanced across the nodes. The development of virtual machine live migration [31] and high-availability cully2008remus further enhanced the capability of a virtualiztion cluster.

As the GPU virtualizaton technology [38, 81, 53] evolves, many virtualization systems are now capable of provisioning and managing GPU resources. The research on efficiently utilizing the heterogeneous resources is expected to be an important

topic for virtualization cluster enhancement.

CHAPTER III

GPU CLUSTER OVERVIEW

Power consumption is one of the major technical constraints in developing high-performance computing (HPC) clusters. Currently, the world's fastest supercomputer Tianhe-2 consumes roughly 18MW of electricity for computation and another 6MW for cooling. That's more power than a town of 10,000 people would typically need. Due to the increasingly tight power constraint, a large amount of research efforts have been devoted to finding a energy-efficient alternative to the traditional HPC clusters. According to the Green500 list ranking supercomputers by energy efficiency, offloading computation from CPU to GPU (Graphics Processing Unit) has become the most effective approach in cutting down power consumption of high performance computing cluster.

During the past decade, GPU has gradually evolved from a specialized processor to a general-purpose parallel accelerator. Thanks to its high-performance many-core architecture, GPU has been successfully used to accelerate a wide variety of applications ranging from media transcoding to scientific research [98, 93]. More importantly, GPU can deliver much higher performance-per-watt than contemporary CPU. On the latest issue of the Green500 list, all the top 10 spots are occupied by GPU clusters[1].

On the other hand, however, the introduction of GPU to the cluster also creates heterogeneity over the computational resource. Unfortunately, the software stacks of most existing GPU clusters are unable to effectively handle such heterogeneity, and are prone to serious underutilization.

[1]The No.2 cluster Suiren uses a proprietary many-core accelerator that is comparable to GPU.

The objective of this research is to improve the throughput of GPU clusters by enhancing resource utilization. In particular, our work is focused on increasing the workload parallelism. In this chapter, we first give a overview of the GPU clusters, followed by studies of the underutilization problem. Our design of enabling cluster-wide GPU sharing and heterogeneous job collocation are then presented. The related works are also discussed.

3.1 System Overview

Existing GPU clusters are designed upon the blueprint of traditional CPU-only HPC cluster, which consists of a large number of identical multi-core computer nodes and a high-performance interconnection. Multiple GPU devices are then plugged into the PCI-E bus of each computer node to form a GPU cluster. A computer node usually has less GPU devices than CPU cores.

In this research, we are using Georgia Tech's Keeneland [125] as the prototype of GPU cluster. This system is initially deployed with 120 HP SL390 computer nodes and a 10Gbps Infiniband interconnect. In each node of Keeneland there are 2 Intel XEON hex-core CPUs and 3 NVIDIA Fermi GPUs. As for the software stack, Keeneland is configured as a batch-processing platform with three major components: the job scheduler PBS, the parallel computing library MPI, and the GPU computing library CUDA [94] (and its open-source counterpart OpenCL [118]).

To use such a platform, the users can write their applications as multi-process jobs, in which every process is composed of interleaved CPU code segments and GPU code segments (GPU kernels). A GPU kernel is essentially a blocking function call, during which the process copies the data into the GPU device, waits, and then copies the output back from GPU. The kernels are statically mapped to local GPU devices and executed in FIFO order. The sibling processes of one job can communicate with each other by calling MPI primitives in the CPU code.

3.2 Underutilized Computational Resource

Although the GPU clusters are designed to deliver high performance with exceptional energy efficiency, the effective throughput of such a system can be compromised by underutilized computational resources.

We monitored Keeneland's PBS system for a period of ten days and gathered a total of 1,669,360 utilization samples, each of which represents the state of one computer node in one minute. According to Keeneland's node allocation policy, there are 3 major states: offline, idle, and exclusively executing a certain job. The four most popular *ppn* (processes per node) settings on Keeneland are $ppn = 1, 3, 6, 12$, since the CPU to GPU ratio is $3 : 12$. The utilization statistics are listed in Table **??**. Among all the busy computer nodes, the average utilization of CPU and GPU are only 47.5% and 29.6%. Since, by default CUDA/OpenCL setting, the processes will actively spin on the CPU while waiting for results from GPU, the effective CPU utilization should be even lower than the percentage value we monitored.

Table 1: Utilization Statistics on Keeneland

Node State	Sample		Utilization		
	Volume	Ratio	CPU	GPU Cyc.	GPU Mem.
$ppn = 1$	96871	5.7%	11.5%	15.6%	4.8%
$ppn = 3$	485582	28.7%	25.2%	34.9%	13.1%
$ppn = 6$	126683	7.5%	43.8%	58.7%	1.7%
$ppn = 12$	414254	24.5%	83.3%	17.9%	4.1%
Idle	417708	24.7%	0.0%	0.0%	0.0%
Offline	128770	7.6%	n/a	n/a	n/a

The fundamental cause of such underutilization problem is the heterogeneity in GPU cluster's computational resources. Although both CPU and GPU are capable of computation, the two are good for different type of workload, due to their distinctive micro architectures. CPU is good for running sequential code and making irregular memory access, while GPU is good for large-size data-parallel workload. To make

a GPU-assisted application run efficiently, the developer has to carefully code the application into separate CPU segment and GPU segment. In many cases, the ratio between such segments is solely dependent on the characteristics of specific workload. For any GPU cluster, however, the physical ratio between CPU cores versus GPU devices is fixed. Consequently, the overall computation capacity of a GPU cluster won't ever be fully utilized, unless the workload-dependent demand can match the underlying heterogeneous CPU/GPU resources.

In practice, several constraints of the prevailing PBS-MPI-CUDA platform further aggravates the underutilization problem. First, the CUDA library requires the computation processes to be statically bound to GPU. All the GPU kernels initiated by a process are executed on the local GPU device that is bound to it. Since there are usually more GPU devices than CPU cores, one GPU device is likely to be shared by multiple processes. Although a GPU can maintain multiple contexts, it can only execute one kernel at a time. If there is contention on certain GPU, all the associated computation processes would stale. Such static mapping between kernels and devices prevents the limited GPU resource from being efficiently shared.

Second, MPI applications are usually designed under the implicit assumption that each software process runs with dedicated resource. Assuming there are M CPU cores available on the node, the PBS system would typically allow no more than M processes being scheduled to that node. In many clusters, the nodes are also exclusively allocated, such that no two jobs would share a computer node. Although such policy can provide better resource isolation, it prevent the system from co-locating complementary workloads to even out the mismatch.

3.2.1 Model Analysis

To study the underutilization problem, we decided to model the system with following abstraction. The GPU cluster is made up of N identical computer nodes, each of

which has M CPU cores and K GPU devices equipped (without loss of generality we assume $M \geq K$). Multiple user jobs can be handled simultaneously on this cluster. Each of the jobs consists of one or more processes, and each of the processes executes a program of multiple iterations of interleaved CPU code segments and GPU kernels. For any process, the execution of CPU code and GPU kernel do not overlap. We denote the ratio of GPU kernel time to the entire execution timespan as GPU computation ratio r_g. To simplify the problem, we assume that the processes have identical workload and do not synchronize with each other.

Based on the typical resource allocation policy of existing GPU clusters, we further assume that a node in this system can host no more than M processes, and can be allocated to no more than one job. A new coming job will be scheduled to run only if enough nodes can be allocated for all of its processes, and then the processes will be evenly scheduled to the allocated nodes. If more than K processes are scheduled to certain node, the GPU kernels of these processes may have to compete for the K GPU devices equipped on the node. The mapping between kernels and devices is static and the service model is FIFO.

For a computer node that hosts p active processes, the CPU utilization u_c and GPU utilization u_g can be derived as:

$$u_c = \begin{cases} \frac{p(1-r_g)}{M} & \text{if } pr_g < K, \\ \frac{p(1-r_g)K}{pr_g M} & \text{otherwise.} \end{cases} \tag{1}$$

$$u_g = \begin{cases} \frac{pr_g}{K} & \text{if } pr_g < K, \\ 100\% & \text{otherwise.} \end{cases} \tag{2}$$

In Figure 2, we use the configuration of Keeneland ($M = 12$, $K = 3$) to demonstrate the relationship between resource utilization and r_g on a single computer node. It can be clearly seen that the utilization of the CPU and GPU are actually in conflict. Since most systems have more CPU cores than GPU devices, the GPU resource

saturates faster if enough processes are scheduled to the node.

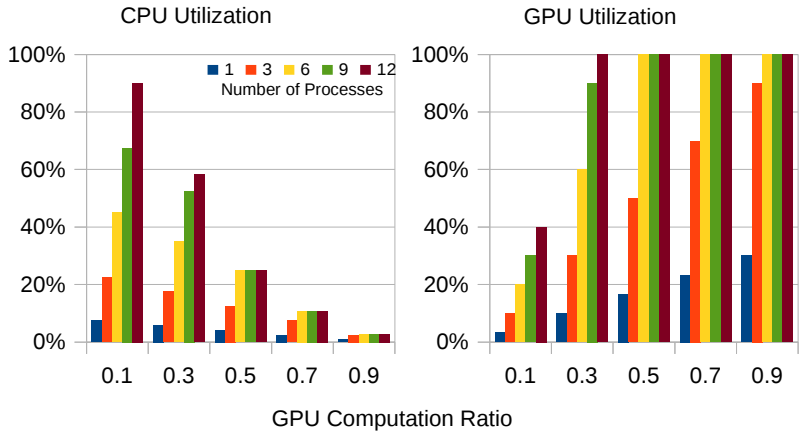

Figure 2: Relationship Between Resource Utilization and GPU Computation Ratio

The result indicates that the underutilization problem would emerge indefinitely in a GPU cluster when the workload pattern of the incoming jobs mismatches the availability of physical resources.

1. *GPU underutilization* would emerge as GPU devices may be idle between kernels, especially when GPUs are used sporadically. Additionally, algorithmic requirements may restrict the application processes to utilize only a subset of the locally available GPUs, thus wasting other GPUs. For example, an application may be designed to use 2 GPUs on each node but is wastefully deployed to a 3-GPU-per-node system.

2. *CPU underutilization* would emerge as CPU cores may be idle waiting for the output of GPU kernel to be returned. This is especially true when the application consists of a large number GPU intensive processes and the allocated computer nodes have limited number of GPU devices. With the static kernel-device mapping, an application process may be stalled by a busy GPU even if there are idling GPU nearby.

3.3 Proposed Solutions

In this study, we are trying to improve the throughput of GPU clusters by enhancing the utilization of heterogeneous computational resources. The fundamental goal is to make the CPU/GPU ratio of workloads match the ratio of the physical resources by lifting the constraints of static kernel-device mapping and exclusive resource allocation.

3.3.1 Dynamic Kernel-Device Mapping

We first evaluate the possibility of balancing the GPU demand across the cluster with remote kernel execution technologies. We argue that the cluster-wide GPU utilization when running mixed workloads can be improved if the GPU kernels can be dynamically mapped to the remote devices that would otherwise be idle (rather than the local devices that are suffering congestion). In particular, our study focuses on the dynamic mapping policies that can adapt to changing workload patterns and network bandwidth.

We demonstrate the performance of the adaptive mapping policies by comparing against native systems (with static local kernel-device mapping). The results show that the dynamic kernel-device mapping outperforms the existing static execution environment in terms of the GPU utilization ratio and the computation throughput, especially for unbalanced mixed workloads.

3.3.2 Heterogeneous Job Collocation

We then evaluate the possibility of improving CPU utilization by smartly co-locating heterogeneous jobs – using extra CPU-only jobs to scavenge the underutilized CPU resource that is created by the GPU-assisted jobs. Since the resources can be over-subscribed when the extra workload are inserted, the main objective of our work is to explore various performance degradation factors in co-locating heterogeneous jobs. Our benchmark studies indicate that the execution time of most co-located jobs won't

be prolonged by more than 5% if the following factors are properly managed: 1) CPU resources isolation, 2) combined CPU load and 3) GPU memory copy bandwidth.

Based on the study, an experimental GPU cluster with collocation-aware job scheduler and resource manager is presented and evaluated. The result shows that job collocation can significantly improve the throughput and utilization when executing mixed CPU-only jobs and GPU-assisted jobs.

3.4 Related Works

3.4.1 Unifying CPU and GPU

An alternative way to optimize the utilization of the heterogeneous resources is to create a unified abstraction and hide the heterogeneity from workload. Several high-level programming libraries are thus developed to unify CPU/GPU computing and to make the workload partition transparent to the programmer. GPUSs [15] uses compiler annotations to define function variants for heterogeneous processing units. Maestro [115] proposes a runtime on top of OpenCL, which enables the workload to be dynamically partitioned and sent into the heterogeneous computational units while maintaining program dependency. MapCG [62] and Merge [84] use the MapReduce model to specify tasks that can be dynamically offloaded to CPUs and GPUs. However, these libraries have strong assumption on the behavior of the application, which makes them less versatile than the combination of PBS-MPI-CUDA/OpenCL and also incompatible to legacy HPC applications.

3.4.2 GPU Virtualization

The research of GPU virtualization is closely related to our work of improving GPU cluster utilization. The early studies on GPU virtualization [81] are focused on enabling concurrent kernel execution by providing a virtualized GPU invocation layer (a daemon process) through which multiple GPU kernels of different application processes can be launched to a single physical device. GViM [53] is another interface-level

solution to virtualize GPU systems. GViM is not designed to access remote GPUs since it can only virtualize GPUs on a standalone computer. Shadowfax [89] is proposed to address the access limit and to support unmodified applications in multiple virtual machines in order to share both local and remote GPUs. Similar to the static designation of native CUDA, all virtual GPUs in Shadowfax need to be manually mapped to a physical GPU, which is unsuitable for managing GPU clusters where user application requests are not known as a priori.

CHAPTER IV

DYNAMIC KERNEL-DEVICE MAPPING ON GPU CLUSTER

In this chapter, we present dynamic kernel-device mapping, which relaxes the static binding between GPU kernels and local GPU devices as in native GPU clusters. We first give the framework of a dynamic mapping enabled system that combines functionalities of remote kernel execution and GPU resource management. The dynamic mapping policies are then designed to balance the cluster-wide utilization of GPUs. The performance of the policies is evaluated with a discrete event based GPU cluster simulator.

4.1 System Framework

The prevailing GPU invocation method is restricted to access the local GPUs - the kernel calls are routinely directed to the GPUs on local computer node only as illustrated in Figure 3(A). With the development of the remote kernel execution libraries such as rCUDA [39] and gVirtus [49], the scope of invocable GPUs is expanded to all the GPUs across the cluster system, as illustrated in Figure 3(B). In these libraries, however, the mapping between the GPU kernel and device is still statically bound.

To relax such static restriction and enhance the cluster-wide GPU utilization, a GPU resource management module (GREMM) is added into the infrastructure, as illustrated in Figure 3(C). Our studies on the dynamic kernel-device mapping policies are based on following framework of a remote kernel execution enabled system:

1. The front-end API library should provide equivalent interface as existing GPU

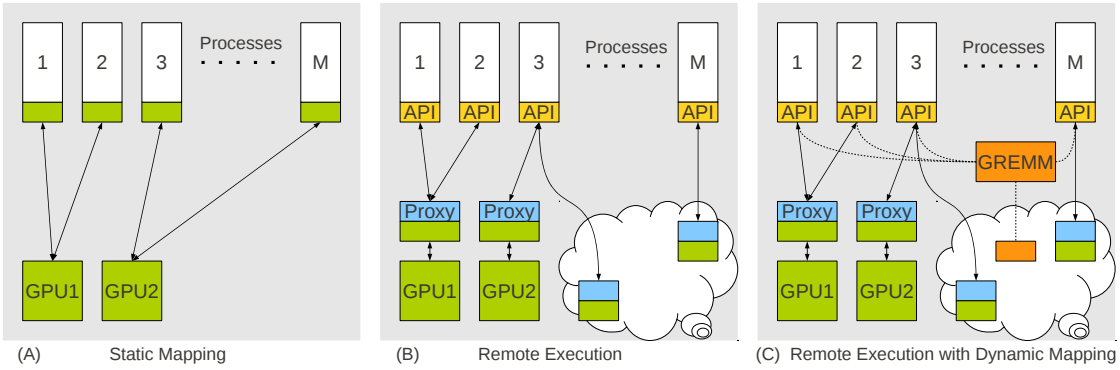

Figure 3: Illustration of GPU Kernel-Device Mapping Models

programming environment such as CUDA or OpenCL, and implement functionalities that communicate to the GREMM. By linking to this library, programmers can write conventional GPU codes for their applications without considering how the GPU kernels are handled. Once the executable is linked with this library instead of the stock one, GPU related functions will be automatically wrapped into network messages and dynamically forwarded to proper proxy.

2. The GPU resource management module is a direct and easy extension of the existing remote kernel execution library. As the broker between GPU API calls and the execution proxies, the GREMM is responsible for making the kernel mapping decisions. A variety of policies can be installed in this module.

3. The GPU execution proxy is the bottom layer of the dynamic mapping framework responsible for the host/device memory copying, kernel launching, and other device control functions. Each proxy controls one local GPU device and communicates with the local and remote API callers. Guided by the GREMM, every GPU task message will eventually be served at a execution proxy.

Although existing GPUs are capable of maintaining multiple contexts, in this work we assume that the application process will explicitly copy back any useful data from GPU after a kernel is finished, so the GPU context becomes volatile and the

process is free to launch new kernels on any GPU. This assumption does potentially underestimate the incentive of reusing same device. However, it resembles a large set of GPU-assisted applications that are based on the computation offload model.

4.2 Dynamic Kernel-Device Mapping Policies

The essential objective of the dynamic mapping policies is to speedup the application process by trading-off between two types of overhead: the queuing overhead and the network overhead. Applications with locally mapped kernels are only subjected to the overhead of queuing for GPU devices. Applications with remotely mapped kernels are further subjected to the overhead of transferring input/output data across network. The intensity of queuing overhead is related to the demand of GPU devices. The intensity of network overhead is proportional to the data volume and network bandwidth. In this section, we are presenting three dynamic policies, in which the overheads are estimated and handled differently.

4.2.1 Global Reservation Policy

In Global Reservation (GR) Policy, a FIFO queue is set up for the GPU cluster. GPU kernels launched by any process will be registered in this queue, which will later be served by a total number of $N \times K$ GPU devices. The actual data transfer occurs directly between the requesting process and the serving GPU, and is not transferred via the queue. Theoretically, if an infinite fast network interconnection is given, the global reservation policy is expected to achieve the best system-wide GPU utilization.

However, the efficiency of this policy is highly sensitive to the network overhead, because the GPU device needs to be reserved while data/kernel is being transferred from a remote node. For the dynamic kernel-device mapping to perform well under environments of varied workload, adaptive policies are then explored.

4.2.2 Adaptive Greedy Policy

Adaptive Greedy (AG) Policy aims to map the kernel call to the GPU device that requires the least total waiting time every time a new kernel call is initiated.

Denote the number of CPUs per node be M, the number of GPUs per node be K, the number of nodes in the system be N, the nominated inter-node bandwidth be B. Denote all the GPUs in the system be G and the set of local GPUs be L, so $|G| = N \times K$ and $|L| = K$ The policy examines every GPU device g in the system, estimates the total waiting time τ_g if the kernel call is mapped to that GPU. The total waiting time τ_g is composed of the queueing delay τ_g^q and the data transfer delay τ_g^d.

$$\tau_g = \tau_g^q + \tau_g^d \tag{3}$$

The queueing delay τ_g^q is estimated by the number of queued kernel calls on that GPU device Q_g and the average execution time of last h kernel calls on that GPU device τ_g^h.

$$\tau_g^q = Q_g \times \tau_g^h \tag{4}$$

The data transfer delay τ_g^d is zero if g is a local GPU device and is estimated by the amount of data transferred from the host node to the remote node D_{out}, the amount of data transferred back from the remote node to the host node D_{in} and the outbound (resp. inbound) bandwidth B_g^{out}(resp. B_g^{in}) if g is a remote device.

$$\tau_g^d = \begin{cases} 0 & \text{if } g \in L \\ \frac{D_{out}}{B_g^{out}} + \frac{D_{in}}{B_g^{in}} & \text{otherwise.} \end{cases} \tag{5}$$

B_g^{out} is estimated by the nominated inter-node bandwidth B and the number of out-bound kernel calls on the host node, namely O_l and the number of in-bound kernel calls on g, namely I_g when the kernel call is to be assigned.

$$B_g^{out} = \frac{B}{\max_{g \in G-L}(O_l, I_g) + 1} \tag{6}$$

B_g^{in} is estimated by the nominated inter-node bandwidth BW and the system-wide average number of queued kernel calls per node. Notice that B_g^{in} is different from B_g^{out} as the bandwidth may change with the progress of the kernel execution.

$$B_g^{in} = \frac{B}{\sum_{g \in G-L} \max(O_g, I_l)/|G|} \tag{7}$$

AG chooses the node g^* with the least total waiting time as the candidate node that the kernel call is to be assigned. The computational complexity of AG is $O(|G|)$.

$$g^* = \arg\min_{g \in G} \tau_g \tag{8}$$

4.2.3 Adaptive Random Policy

Adaptive Random (AR) Policy is a randomized policy. It tries to construct and maintain a table which records the probability that a particular GPU device should be chosen to serve the kernel call. It assigns the kernel call based on the probabilities in the maintained table. It resorts to the GPU driver to handle the contention for the GPU device on a particular node if there is any.

The probability of being chosen is calculated based on a weight table that is associated with the system wide GPU availability. In this table, each GPU device is assigned a weight indicating the relative probability of being chosen.

Denote the weight of a remote idle resp. busy) GPU device be w_{ri} (resp. w_{rb}), the weight of a local idle (resp. busy) GPU device be w_{li} (resp. w_{lb}).

Assume that the inertia towards choosing the busy GPU devices over the idle ones is characterized by a 'penalty' factor $\alpha(< 1)$, and that the preference towards choosing the local GPU devices over the remote ones is characterized by a 'bonus' factor $\beta(> 1)$. Hence we have

$$\alpha = \frac{w_{lb}}{w_{li}} = \frac{w_{rb}}{w_{ri}}, \tag{9}$$

$$\beta = \frac{w_{lb}}{w_{rb}} = \frac{w_{li}}{w_{ri}}. \tag{10}$$

Without loss of generality, if we set the $w_{ri} = 1$, then $w_{li} = \beta$, $w_{rb} = \alpha$, $w_{lb} = \alpha\beta$.

The 'penalty' factor α can be quantified by the average execution time of received kernel calls τ_g^h and the node configuration of the host node.

$$\alpha = \frac{M}{K} \times \frac{1}{\tau_g^h} \tag{11}$$

The design philosophy of α is that the relative probability ratio of choosing a busy node over choosing an idle node should be proportional to the relative ratio of the time ticks that a node is idle, and that the more GPU devices reside on a host node, the less chance the kernel calls should be assigned to remote nodes.

The 'bonus' factor β can be quantified by the amount of transferred data D, the average execution time of received kernel calls τ_g^h and the number of nodes in the system N. The design philosophy of β is that the higher the ratio of the communication time to the computation time, or the higher the data consumption rate, or the more nodes in the system, the more chance the local nodes are favored over the remote nodes.

$$\beta = \left(\frac{D/B}{\tau_g^h}\right) \times \left(\frac{D}{\tau_g^h}\right) \times N = \left(\frac{D}{\tau_g^h}\right)^2 \times \frac{N}{B} \tag{12}$$

The computational complexity of AR is also $O(|G|)$ while the information it needs to keep is less than AG. When a GPU kernel request arrives, GREMM makes the randomized assignment decisions based on the weights in this table.

More sophisticated mapping policies can be designed, for example, to explore execution history of the system and to accept users' hint about the pattern of their jobs. In this work, we focus on the benefits of dynamic GPU kernel-device mapping and its effectiveness under different workload and system conditions. Greater performance improvement is expected when more advanced mapping policies are adopted.

4.3 Evaluation Setup

In this section we develop a discrete event based simulator to study the runtime behavior of large-scale GPU clusters. The performance of dynamic kernel-device mapping strategies is then verified through extensive simulations.

It is desirable to evaluate the dynamic kernel-device mapping policies in a large-scale production GPU using real benchmark workloads. But because GPU cluster computing is an emerging field, there is well-established workload traces for this type of systems. To address this issue, we synthesized our workload mix, which is based on the characteristics of real GPU-assisted applications.

4.3.1 Experimental Setup

The four GPU mapping policies tested are: 1) ST, the static kernel-device mapping policy; 2) GR, the global reservation policy; 3) AR, the adaptive random mapping policy with $h = 10$; and 4) AG, the adaptive greedy mapping policy with $h = 10$. The ST policy, as our baseline, is the conventional policy in GPU execution environment which shows the GPU utilization of the native system without remote execution or dynamic mapping. GR, AR, and AG are dynamic kernel-device mapping policies.

The two major performance metrics evaluated are GPU Utilization Rate and Mean Waiting Time. GPU Utilization Rate is the ratio of the GPU busy time to the total GPU time available. This rate directly reflects the utilization efficiency of the entire GPU cluster. The Mean Waiting Time measures the average time that a GPU kernel spends on data transfer and queuing for GPU devices. It reflects the average overhead for each kernel execution.

4.3.2 GPU Cluster Simulator

The simulated cluster consists of N computing nodes. Each node consists of M CPU cores, K GPU devices, and a full-duplex network interface card (NIC) with max bandwidth B. The CPU cores are characterized by the processing capabilities. GPU devices are also characterized by their processing capabilities. The latency of remote execution API functions is modeled based-on data observed in previous researches [39, 53, 89].

The NIC on each node has two independent ports: the inbound port and the

outbound port. A max bandwidth B is enforced on each port. Once any data is to be transmitted from one node to another, a connection will be established from the outbound NIC port of the source to the inbound NIC port of the sink. Concurrent connections on a single port share the port's bandwidth evenly. However, the effective bandwidth of a connection is limited by the busier one of the two participating ports. So, if any one of the concurrent connections fails to fully utilize its share, the remaining bandwidth will be utilized by other concurrent connections. According to this scheme, the system-wide bandwidth allocation changes when a new connection is established or a current connection is completed. We adopt this simplified network model as our focus is on the impact of network transfer overhead, rather than on how the overheads are generated. Therefore, the detail characteristics of a typical network such as the topology and the routing are not taken into account in this work.

Unless explicitly noted later, the cluster in following simulations is configured as $N = 24$, $M = 12$, $K = 3$. The bandwidth is set to $B = 100KB/ms$ for 1 Gbps Ethernet (GbE) and $B = 1000KB/ms$ for 10 Gbps InfiniBand (IB). The simulated time span is $10,000,000ms$.

4.3.3 Generation of Workload Traces

The input to the simulator is the workload trace, which is organized as groups of consecutive *program segments*. Each group is associated with one software process. We characterize a CPU segment by the amount of time T_c delayed on the CPU core, and a GPU kernel by three parameters: the amount of data D_u uploaded to the GPU device; the amount of execution time T_g on GPU device; and the amount of data D_d downloaded from the GPU device to host process.

The workload used for the evaluation of the dynamic mapping framework is generated based on following assumptions:

- Each process executes CPU and GPU segments alternatively.

- The execution time of each segment is random variable of exponential distribution with parameter $\lambda = 1/T_c$, $\mu = 1/T_g$ respectively.
- The size of the input and output data sets of a GPU kernel is proportional to the kernel execution time.

For example, a process P generated with parameters $T_g = 2250ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$, and $T_c = 750ms$ will have the following characteristics: the average length of GPU kernel is $2250ms$; the average data uploaded to GPU each time is $22500KB$; the average data downloaded from GPU each time is $11250KB$; and the average time spent on CPU before next GPU kernel launch is $750ms$.

Since the policies are designed to address unbalanced GPU utilizations of concurrent GPU workloads in a GPU cluster. The traces we used are mixed combinations of a heavy-GPU application and a light-GPU application. We assume that the system runs these two applications with full capacity: there are n_i (assuming n_i is a multiple of $\frac{M}{K}$) processes in Workload i, and $n_1 + n_2 = N \times M$. In such case, the GPUs in the system are subject to the different computation intensity. The benefit of routing a kernel from a stressed node to an idle remote node can potentially overweigh the extra overhead of network transfer. Our analysis can be extended to scenarios with more applications.

4.4 Simulation Result

4.4.1 Workload Mix

Our first set of experiments examines a set of mixed workloads. Traces W_1 to W_5 are synthesized from two client applications submitted to the cluster. Client H's application consists of kernels with heavy GPU usage (with $T_c = 750ms$, $T_g = 2250ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$) and client L's application consists of kernels with light GPU usage (with $T_c = 2250ms$, $T_g = 750ms$, $D_u = 10 \times T_g$, $D_d = 0.5 \times D_u$). The five traces are synthesized to represent the mix of two workloads with different GPU

demands. The process population ratio of H/L is 24/0 in W_1, 18/6 in W_2, 12/12 in W_3, 6/18 in W_4, and 0/24 in W_5.

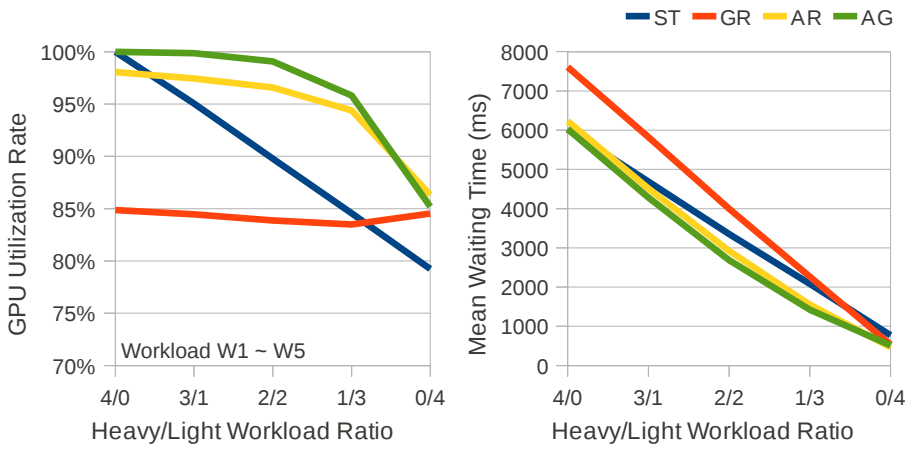

Figure 4: Impact of Workload Mix

The system is simulated with the network bandwidth set to $100KB/ms$ (GbE). As shown in the left subplot of Figure 4, the system-wide GPU utilization rate can be improved by dynamic mapping policies in most of the cases. Since there are under-utilized GPU devices on the nodes, transferring GPU kernels from heavily occupied local devices to remote idle devices is beneficial. It is worth noting that significant improvement can be observed for the adaptive policies even for such low bandwidth network. This indicates that the dynamic kernel-device mapping is particularly useful for mixed workloads that have different GPU demands. Meanwhile, the mean waiting time is also improved as is shown in the right subplot of Figure 4.

Figure 5 shows the number of completed GPU kernels under different policies on Traces W_1 to W_5. Taking W_3 as an example, the conventional ST policy finishes about $12K$ kernels for client H and about $28K$ kernels for client L in the simulated time span. When the dynamic policies are applied, the overall system-wide GPU utilization rate is improved. It is also interesting to note that client L is affected by the other policies, i.e. client L completes less kernels if remote GPU mapping is allowed. This is because the GPUs previously dedicated to client L are now executing client H's kernels too.

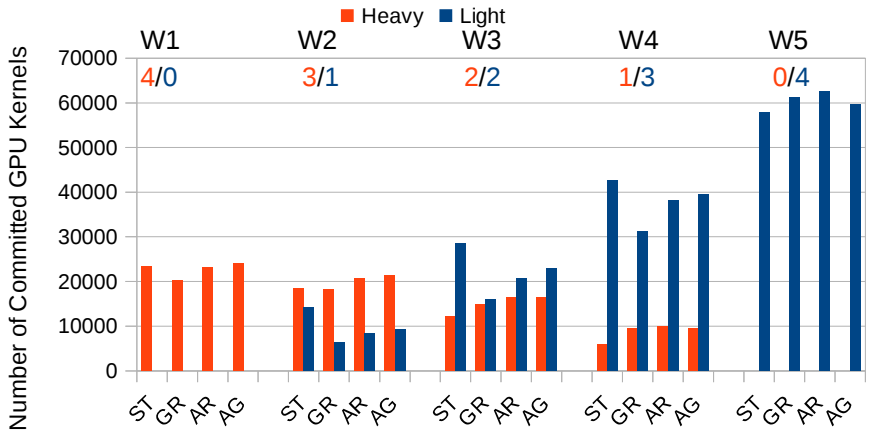

Figure 5: Number of Completed GPU Kernels with Different Policies

This set of experiments suggest that if certain client's application is mission-critical, it is desirable to exclude other applications from utilizing its GPU devices, even though this will reduce the GPU utilization rate of the system. We plan to investigate the prioritized policy in our future study.

4.4.2 Load Balance

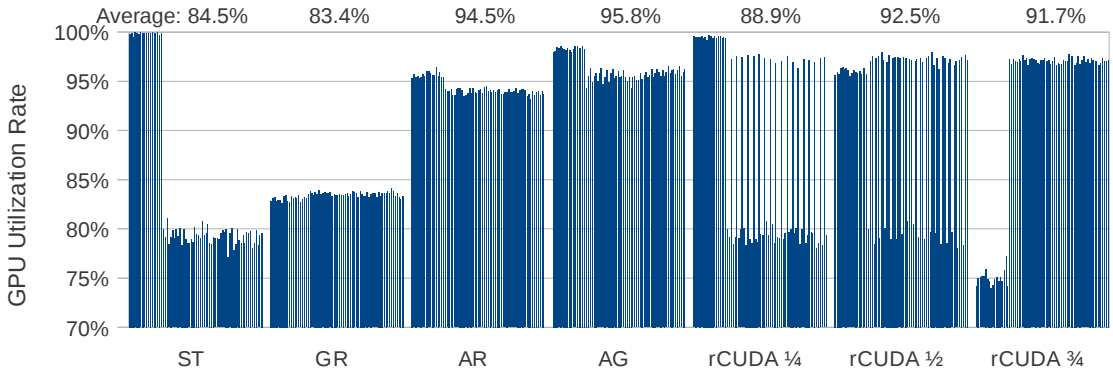

Figure 6: Detailed GPU Utilization of the Cluster with Static and Dynamic Policies

Figure 6 lists the detailed GPU utilization of the cluster with different policies on the mixed workload trace W_4, since W_4 is a very good example to demonstrate the performance improvement of the dynamic kernel-device mapping. The bandwidth is set to $100KB/ms$ here and in the following experiments as well. It shows that the utilization with ST is negatively affected by the unbalanced node utilization. The GR

policy is capable of balancing GPU utilization. The AR and AG policy outperforms the other policies for this set of experiments.

As mentioned in the background section, techniques such as rCUDA allow a process to send all its GPU kernels to a statically designated remote node, but they do not support run-time kernel-device mapping. For fair comparison, we tested three static schedulers for rCUDA on workload W_4: 1/4, 1/2, and 3/4 of the client H's GPU kernels were directed to client L's GPUs. The results show that when 1/2 of client H's processes can use rCUDA, the system achieves GPU utilization rate of 92.5%, which is still worse than the performance of the dynamic mapping policies. Nevertheless, the results also demonstrate the difficulty in optimizing the performance by the static scheduler of rCUDA: ratios 1/4 and 3/4 are less efficient, finding the better ratio of 1/2 is non-trivial. Furthermore, since the rCUDA mapping decision needs to be made before launching user applications, it is infeasible to use rCUDA for actual high-performance applications since there does not exist a single static mapping policy that will suit all kinds of workloads.

4.4.3 Workload Intensity

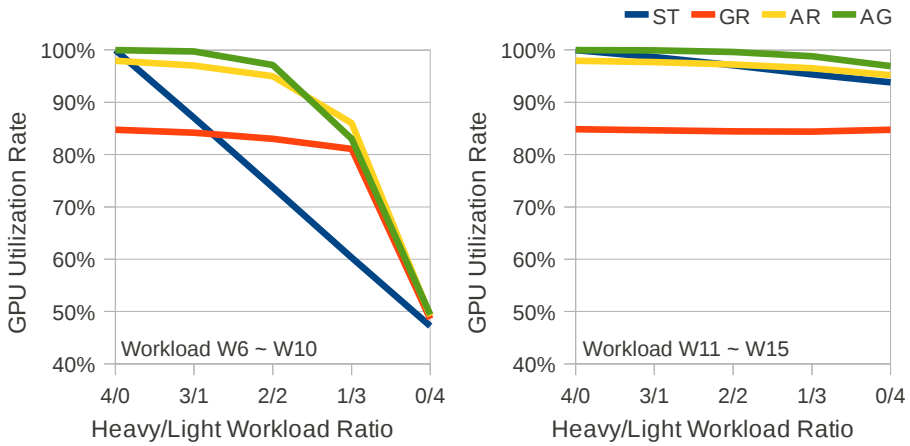

Figure 7: Impact of GPU Utilization Intensity

The impact on the GPU utilization intensity is demonstrated in Figure 7. Two

new groups of workloads (W_6-W_{10} and W_{11}-W_{15}) are used in the experiment. The generating parameters of these workloads are the same as that of W_1 to W_5, except that the (T_c, T_g) of light workload is set to $(2625, 325)$ in W_6-W_{10} and $(1815, 1125)$ in W_{11}-W_{15}. As the results show, the dynamic policies are significantly effective only if enough underutilized GPUs exist. In the lighter group (W_6-W_{10}), up to 26% improvement can be observed, but in the heavier group (W_{11}-W_{15}) the improvement is limited by the existence of over-utilized GPUs.

4.4.4 Network Overhead

In this experiment, we examine the sensitivity of the policies to the network bandwidth and the data/computation ratio of the GPU kernels, which are two key factors that affect the network transfer overheads introduced by the remote execution of GPU kernels.

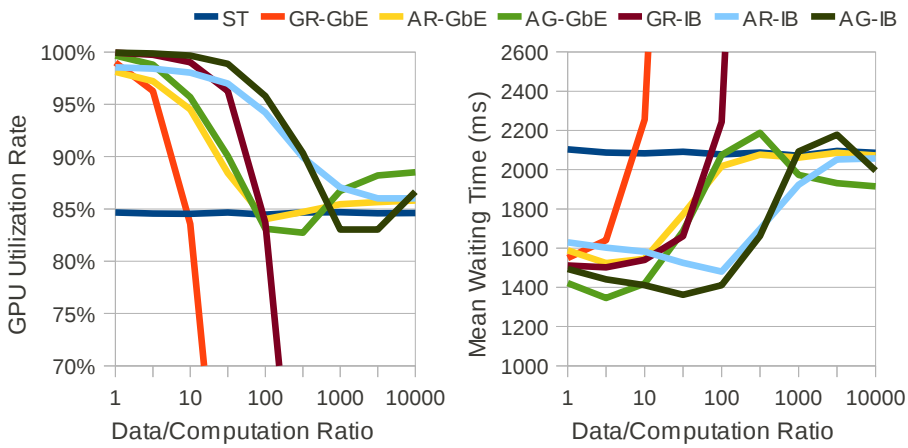

Figure 8: Impact of Network Bandwidth

Figure 8 shows the system-wide GPU utilization of different policies and the underlying interconnect with varied D_u/T_g (data/computation ratio). Here D_u of the workload W_4 is sampled exponentially from $D_u = 1 \times T_g$ to $D_u = 10000 \times T_g$. As D_u/T_g increases, the overhead of remote-execution increases, which negatively affects the performance of the dynamic mapping policies (and especially of the GR policy).

This indicates that the amount of transferred data or the network bandwidth plays an important role in making dynamic mapping policies effective and efficient. However, the performance of AR and AG can still be as good as ST when the ratio is extremely high, thanks to the adaptation mechanism.

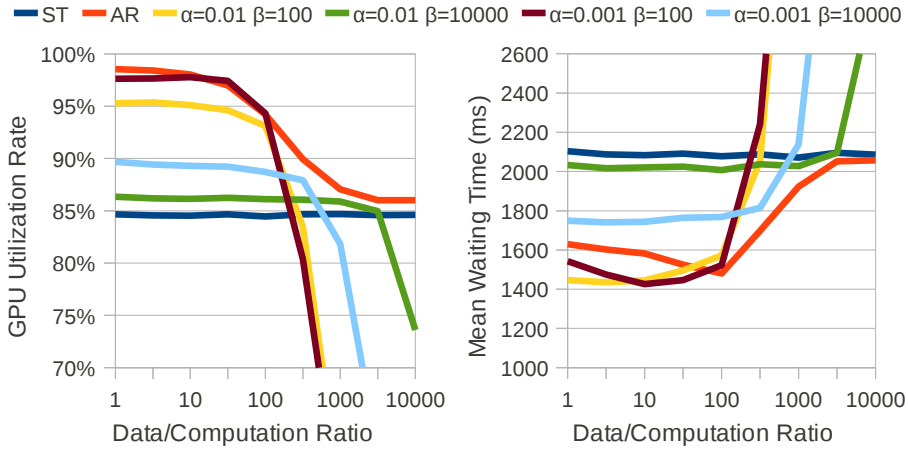

Figure 9: Benefit of Adaptation Mechanism

The benefit of the adaptation mechanism can be clearly demonstrated with Figure 9. In this experiment, we explicitly assign several fixed values to α and β, and compare these fixed-weight random policies to AR. The result reveals that the fixed-weight may favor either the low data/computation ratio workload or the high data/computation workload. Only the adaptive-weight in AR can track the best performance over the entire range of data/computation ratio.

4.4.5 Scalability

The scalability of the dynamic mapping policies is evaluated in the following two experiments: scalability with respect to the number of GPUs per node, and with respect to the number of nodes. Trace W_4 is used for the first set of experiments. For the second set of experiments, the four tested traces are half-sized, normal-sized, double-sized, and quadruple-sized versions of W_4. The network bandwidth is set to

$100KB/ms$. The GR policy is excluded in this experiment due to its poor performance over lower-bandwidth network.

In the experiments, we observe higher possibility of underutilization by static mapping when more GPUs are installed in the cluster. In such cases, the necessity of an efficient GPU resource management policy becomes more significant.

The values reported in Figure 10 are the GPU utilization rate margin of the dynamic mapping policies over the ST policy. According to the results, both AR and AG exhibit good scalability over the number of GPUs per node. The AR policy also exhibits good scalability over the number of nodes. However, the AG policy doesn't scale well with the number of nodes. The key reason is that estimating the delay times in a larger-scale system becomes harder and less accurate. The larger amount of collaborative communication incurred during AG's decision making process also impairs its scalability over the number of nodes.

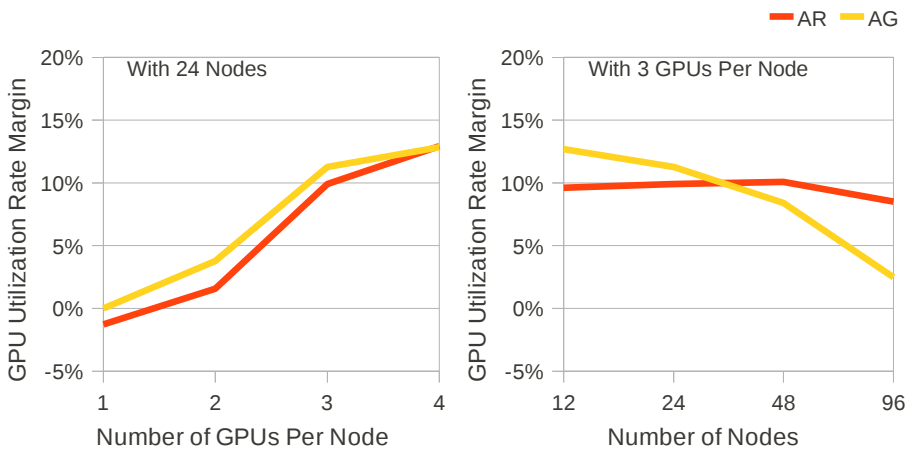

Figure 10: Scalability of Dynamic Mapping Policies over Static Mapping

Since both AG and AR rely on certain amount of global information to make scheduling decisions, their performance could be significantly compromised if the system scales up to thousands of nodes. To accommodate such large systems, one effective way is to group the nodes into subsets and schedule remote GPU accesses within each subset. In future researches, an alternative policy relying on distributed

information and local estimation will be studied.

4.4.6 Design Choices for the AR Policy

This set of experiments evaluates the performance of AR policy over certain design choices. As demonstrated in the previous experiments, AR is a balanced policy with several distinct advantages. One important choice in the implementation of this policy is how to maintain the distributed table about the GPU status. Real-time update is less desirable since it may incur extra network overhead. On the other hand, if the table is updated less frequently, outdated information may be used for GPU kernel/device mapping. We define a case to be false positive if an idle GPU is identified as busy, and false negative if a busy GPU is identified as idle. We used traces W_1 to W_5 to evaluate the performance of AR policy over different false positive ratio/false negative ratio. The values reported in Figure 11 are the GPU utilization rate margin of AR over ST. As shown in the figure, the performance is more sensitive to the false negative ratio than the false positive ratio. This implies that the status should be updated as soon as possible when a certain GPU becomes busy and the update is less urgent when a GPU becomes idle if the performance of the AR policy is valued.

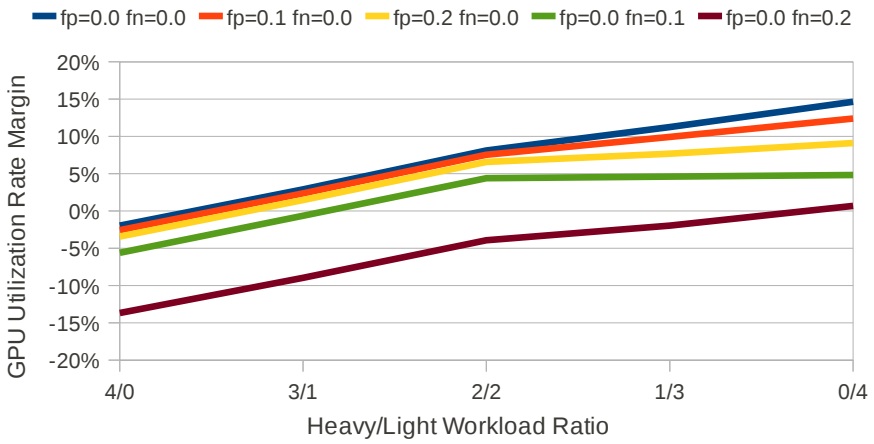

Figure 11: Impact of False Positive and False Negative Ratio on AR

4.4.7 Summary

The simulation studies show that a dynamic kernel-device mapping enabled GPU cluster is capable of delivering higher utilization ratio and computation throughput than the native static mapping based clusters, especially over unbalanced mixed workloads. The simulation also confirms that the communication intensive workload can cause noticeable performance degradation for the remote GPU kernel execution. With the help of adaptive policies (AR and AG), the dynamic mapping enabled systems can effectively fall back to the local execution method and avoid the performance degradation when unsuitable workloads are detected.

4.5 Related Works

The topic of GPU device sharing has been explored in two dimensions: intra-node and inter-node. The researches on intra-node GPU sharing focus on the efficiency of running multiple kernels concurrently on single device. Due to the limitation that many existing GPU devices can only run kernels in FIFO order, the first attempt [50] is to pack the source code of multiple kernels together before sending them onto the device for execution. The native support of concurrent kernel execution becomes available since the release of Nvidia Kepler GPU. Based on this new feature, several new mechanisms [82, 107] have been proposed to enable space-sharing and time-sharing of single GPU device. The study of Kernelet [144] further improves the efficiency of concurrent kernel execution by integrating the source code modification technique with optimized scheduling mechanism.

The inter-node GPU sharing studies, on the other hand, focus on enabling remote GPU kernel execution. As a way to resolve the scarcity of GPU devices in many computer systems, remote GPU sharing has attracted intensive research attention. rCUDA [39] is proposed to enable the computer nodes not equipped with local GPUs to access the remote GPUs hosted on remote computer nodes. It employs

API remoting technique to reroute the GPU calls to a remote computer node. With rCUDA, the remote GPUs are statically specified in a configuration file on the requesting node. rCUDA works between a pair of designated nodes, and is particularly useful in a cluster environment where only a few nodes are equipped with GPUs. In such settings, rCUDA allows other non-GPU nodes to execute their GPU kernels on the GPU-equipped nodes, but the kernels in rCUDA- ased system remains statically bound to devices, since the users have to hard-code the remote rCUDA server IP into their application. rCUDA is not designed to manage GPUs in an GPU cluster. The capability of remote GPU kernel execution is also explored in several other projects such as SnuCL [72], MGP [18], and gVirtuS [49].

The idea of managing cluster-wide GPU resources with a remote-execution framework has been studied in several projects. The work in [102] proposed a interference-driven job scheduling system, where multiple jobs can share the same GPU if GPU memory constraint is not broken, and the CPU and GPU components of one process are scheduled on the same node. The major difference to our design is that their system relies on the prior knowledge gained from static profiling to prevent resource congestion, while we are using adaptive policies to balance the utilization. An alternative GPU resource management framework was proposed in [111], where a dedicated execution proxy is created for each application process. Although such 1:1 design offers high fault tolerance and security, it is less efficient than creating one execution proxy for each node. The research in [110] proposed a preemption-based runtime to efficiently schedule multi-process applications on GPU clusters. Different from previous studies, this proposal uses application's synchronization point as a chance to schedule/deschedule the processes, such that the utilization of the associated GPU device can be altered. However, due to its focus on single application, this work is unable to address the ultimate workload mismatch problem.

CHAPTER V

HETEROGENEOUS JOB COLLOCATION ON GPU CLUSTER

In this chapter, we evaluate the possibility of co-locating CPU-only jobs with GPU-assisted jobs on a GPU cluster to improve the utilization of CPU resources. We first construct a MPI/CUDA-based workload pool with real benchmark applications, and then study the collocation performance against several system-related and job-related factors. Based on the study, an experimental collocation-enabled GPU cluster is implemented to demonstrate how job collocation can enhance the system's utilization as well as throughput.

5.1 Performance Studies of Job co-locating

Intuitively, MPI-based high-performance applications tend to perform worse when co-located with another job that competes over the resources. Whether job collocation can improve the throughput of the entire system depends on the trade off between the CPU cycles scavenged and the overhead incurred. In this section, we evaluate and model how the jobs are slowed down in co-located environment than in dedicated environment. The experiments focus on those key factors that compromises the performance of co-located jobs, which are resource isolation, combined CPU load, and cudaMemcpy bandwidth.

5.1.1 Experimental System Platform

The experimental platform we used is a 4-node cluster with two Intel Xeon X5675 hex-core CPUs, three Nvidia M2070 GPUs, and 24GB of memory installed on each node. All the nodes are connected with a Gigabit Ethernet switch. To minimize the impact

of CPU's specific micro architecture, the Intel SMT and Turbo Boost features are turned off. Our software environment consists of CentOS 6.4, GCC 4.4.6, Openmpi 1.6, CUDA 4.2.9, Torque 2.5.12, and Maui 3.3.1.

5.1.2 Workload Characteristics

Our study on co-locating performance involves jobs of two different categories of workload: CPU-only and GPU-assisted. A group of 15 MPI benchmarks are selected from the NAS Parallel Benchmark (NPB) [16] 3.3.1 as our CPU-only workload, and a group of 32 MPI+CUDA benchmarks are selected from the Scalable HeterOgeneous Computing (SHOC) Benchmark [32] 1.1.4 as our GPU-assisted workload. By combining the two groups, a total of 480 different job combinations are constructed. For conciseness, the two categories of workload are referred to as CPU workload and GPU workload in the rest of the chapter.

The MPI benchmarks we used can be broken down into five basic types: 1) Conjugate Gradient (CG), irregular memory access and communication; 2) Embarrassingly Parallel (EP), minimal communication; 3) Discrete 3D fast Fourier Transform (FT), all-to-all communication; 4) Multi-Grid (MG) on a sequence of meshes, memory intensive; and 5) Integer Sort (IS), random memory access. All these NPB benchmarks are configured with different $Class$ options (which identify problem size) and $NProcs$ options (which identify number of processes). Their baseline execution time on our experimental platform without co-located workload ranges from 2.72 to 84.5 seconds, as listed in Table 2.

Our 32 MPI-CUDA benchmarks can be broken down into eight types: 1) 1D Fast Fourier Transform (FFT), 2) N-body pairwise computation from molecular dynamics (MD), 3) Reduction, 4) Exclusive parallel prefix sum (Scan), 5) General matrix multiplication (SGEMM), 6) Radix sort on integer key-value pair (Sort), 7) Sparse matrix

Table 2: Characteristics of CPU Workload

Name	Class Opt.	NProcs Opt.	Execution Time (s)
CG	B	2 4 8	48.07 27.06 14.67
EP	B	2 4 8	46.56 23.11 11.51
FT	B	2 4 8	49.77 26.656 15.48
IS	C	2 4 8	8.16 4.31 2.72
MG	C	2 4 8	84.50 44.48 22.69

vector multiplication (Spmv), and 8) Streamed Triad. All these benchmarks are configured as 12 processes and single precision (if not otherwise stated). The benchmarks are also configured as embarrassingly parallel on MPI level and as true parallel on CUDA level, such that the MPI processes are executing independently over the same copy of code. Three baseline characteristics (execution time, CPU load, cudaMemcpy bandwidth) are profiled on our platform. The CPU load represents the actual CPU utilization of the benchmark with its 12 processes running concurrently. If the CPUs are fully utilized, the value of CPU load should be greater than or equal to 12. The cudaMemcpy bandwidth is the per process average bandwidth of bi-directional cudaMemcpy traffic, which represents the GPU memory demands of the benchmark. The profiling results, listed in Table 3, show that these benchmarks cover a wide spectrum of GPU-assisted workload patterns. The detailed *Size* and *Iterations* options that we used to configure each benchmark can also be found in the table.

In order to facilitate our experiments, following modifications are applied to the

Table 3: Characteristics of GPU Workload

Name	Size Opt.	Iteration Opt.	Execution Time (s)	CPU Load	cudaMemcpy bandwidth (MB/s)
FFT	4	1	6.50	2.22	39.43
		120	18.47	1.19	13.86
		240	30.60	0.72	8.36
		360	42.64	0.54	6.00
MD	1	2000	7.91	9.87	0.81
		4000	10.92	7.69	0.58
		10000	20.77	4.83	0.31
		30000	55.68	2.82	0.11
Reduction	3	1	0.90	1.17	35.41
		10	4.51	0.70	70.98
		20	8.86	0.61	72.26
		30	13.15	0.59	73.03
Scan	2	1	0.44	1.04	36.72
		4	1.42	0.79	28.10
		8	2.79	0.65	25.80
		16	5.52	0.57	24.62
SGEMM	3	1	1.20	2.98	53.19
		4	4.37	1.26	36.63
		8	8.58	0.75	33.56
		20	21.21	0.35	31.68
Sort	4	1	2.74	1.43	140.00
		8	8.81	1.57	348.81
		16	16.29	1.54	377.08
		24	23.77	1.54	387.77
Spmv	4	1	6.69	5.99	13.50
		4	18.37	2.60	4.96
		8	33.19	1.50	2.78
		16	62.10	0.86	1.53
Triad	4	1	0.75	5.05	576.64
		10	7.07	4.94	611.14
		20	14.30	4.92	604.20
		30	21.65	4.99	598.48

SHOC benchmark code : 1) Assigning the ith GPU device to the host processes whose rank mod GPU count equals i; 2) Explicitly setting CUDA device flag and event flag as BlockingSync to make sure that host CPU process does not busy spin when waiting for GPU; 3) Wrapping the RunBenchmark function into for-loops to enable representative running of benchmark without reinitialize GPU devices.

5.1.3 Performance Metric

In the experiments, we focused on the performance metric of Slowdown Ratio, which is defined as:

$$\frac{(co-locatedExecutionTime - BaselineExecutionTime)}{BaselineExecutionTime}$$

. If both co-located CPU and GPU jobs have small slowdown ratio (e.g. less than 5%), it means that this job combination is able to effectively utilize the system resources and leverage the system throughput. The slowdown ratio can be affected by many conditions. In the following sections, different performance compromising factors are evaluated against this ratio.

5.1.4 Performance Compromising Factors

Several other job characteristics (memory footprint, data precision, GPU kernel launch frequency) and system characteristics (I/O rate, hardware/software IRQ frequency) have also been evaluated, even though we didn't observe significant correlation between these characteristics and the slowdown in co-located jobs. Due the incompleteness of our experiment, however, it is possible that additional factors exist. Another issue involving the study is that the workload didn't cover the possible co-scheduling slowdown when the processes in one job have dependencies among each other.

5.1.4.1 Resource Isolation

When a CPU job and a GPU job are co-located on the same node, the CPU cores on that node can be managed in two ways: shared or partitioned. For the shared mode, both jobs can access any of the cores, and the contentions are handled by Linux's default process scheduling algorithm. For the partitioned mode, either job will have its dedicated subset of cores, which are enforced explicitly with CPU affinity settings. Since in typical CPU jobs every process is able to fully utilize the CPU resource, we

will dedicate equal amount of cores to these CPU processes and the remaining cores to GPU job.

We tested the 480 job combinations separately with these two modes and calculated the difference in execution time. The results are listed in Figure 12. Each blue bar in the figure represents the CPU job slowdown ratio of shared mode over partitioned mode in one specific combination, and each red bar represents the GPU job slowdown ratio in the same combination. To make the figure clearer, the bars are sorted respectively by job combination's CPU load. The figure shows that the CPU job of most job combinations takes longer to finish in shared mode. For the GPU jobs, if the combined CPU load is less than 12, both modes have some losses and gains. In general, co-located jobs perform better if the resources are partitioned. The following experiments are also focused on this mode.

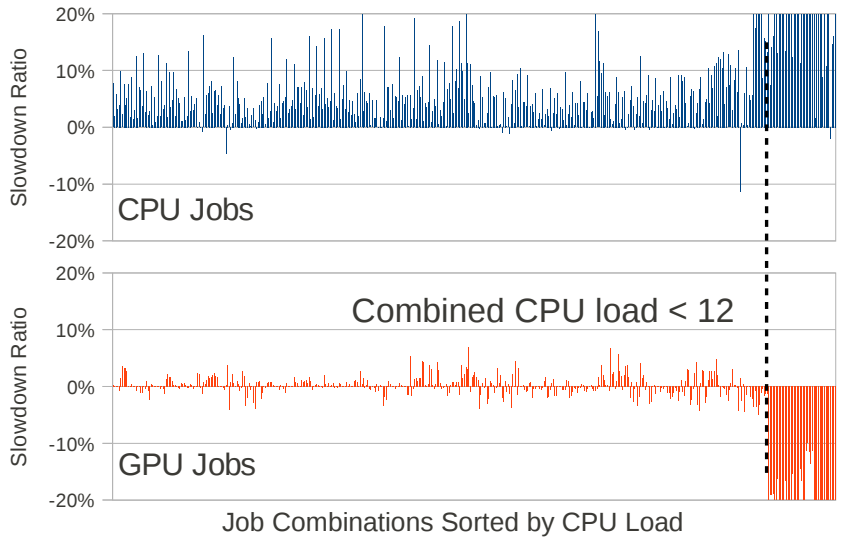

Figure 12: Slowdown of Shared Mode Over Partitioned Mode

5.1.4.2 Combined CPU Load

Figure 13 lists the CPU job slowdowns and GPU job slowdowns of the 480 job combinations over their baseline execution time. Intuitively, both jobs take longer to finish as the combined CPU load increases. However, the performance of GPU jobs

is less vulnerable than CPU jobs when co-located. There are two reasons for the such difference: 1) the host processes of GPU jobs in most SHOC benchmarks are embarrassingly parallel, and 2) the performance of GPU kernels is independent of the performance of its host process. It can also be noticed in the figure that the slowdown in both jobs are much less significant if the combined load is less than 12 (the number of CPU cores per node). When the CPUs become over-utilized, the performance of both co-located jobs decreases dramatically.

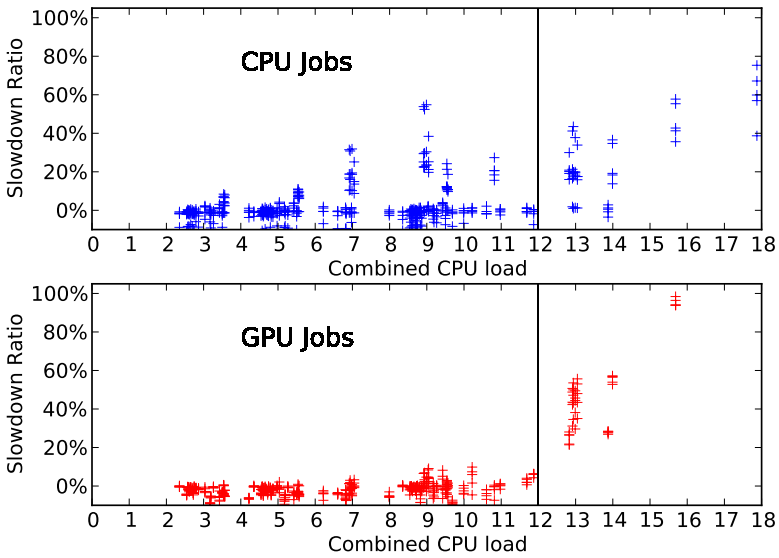

Figure 13: Slowdown of Jobs Against Combined CPU Load

The distribution of performance with combined load ranging from 0 to 12 is listed in Figure 14. It shows that the slowdown of all GPU jobs and of 88% of the CPU jobs are less than 10%. According to this result, if the CPUs are not persistently over utilized, co-locating CPU job can serve as an efficient way to scavenge the CPU resources.

5.1.4.3 cudaMemcpy Bandwidth

As shown in Figure 13, CPU jobs in the majority of the combinations have small slowdown ratio if combined load is less than 12. However, there are some incidents

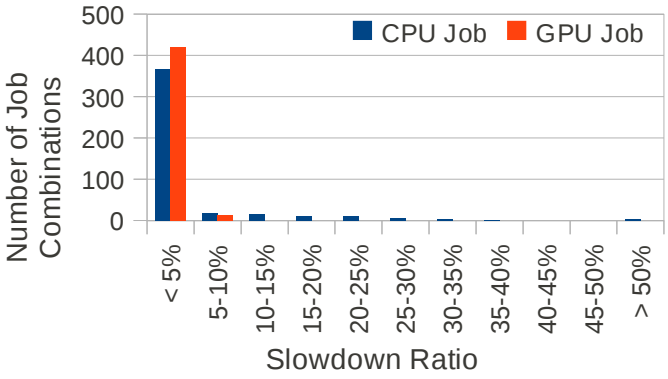

Figure 14: Distribution of Jobs over Slowdown Ratio

when CPU jobs are slowed down even if CPU resources are abundant. In Figure 15, the same groups of results are plotted against CPU load and cudaMemcpy bandwidth respectively, and those results relating to the same type of CPU benchmark are marked with unique color. This figure demonstrates that the slowdown ratio has clear correlation with both the GPU job's cudaMemcpy bandwidth and the CPU job's type. In general, the cudaMemcpy bandwidth exhibits a strong exponential impact over the performance, while different types of CPU jobs have different sensitivity to this impact.

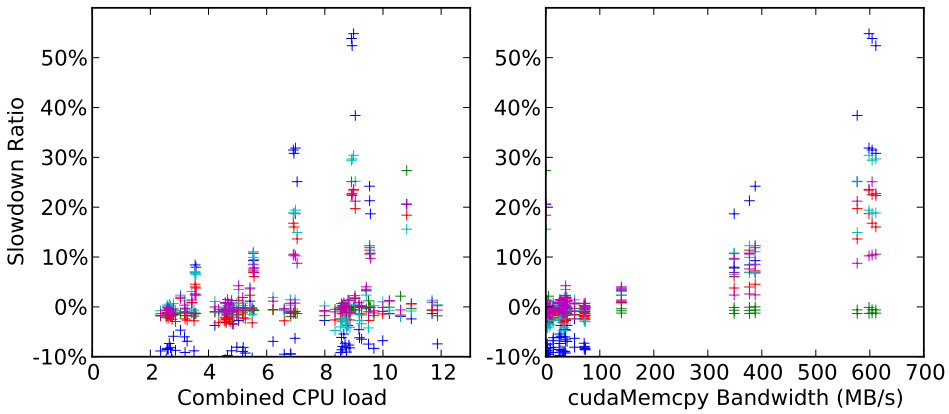

Figure 15: Slowdown of CPU Jobs Against Combined CPU Load and cudaMemcpy Bandwidth

We developed the following regression model to link the slowdown ratio with

multiple parameters of the impact.

$$S = P \times b \times e^{a \times B}.$$

In this model, S is the slowdown ratio, P is the process number of the CPU job, B is the cudaMemcpy bandwidth of the GPU job, a and b are the two coefficients. The regression procedures are illustrated in Figure 16, in which the experiment results related to CG type CPU benchmark are used as samples. The three colors used in this figure represent three process number (2, 4, and 8) of CPU job. The first step is to remove all the results with a small bandwidth (e.g. less than 100 MB/s) since the bandwidth is a minor performance compromising factor in those cases. Then the slowdown ratios are divided by the process number. The divided values are used as samples to fit an exponential curve. Finally, multiply the process number back into coefficients to get the final curves. With these procedures, we can get a pair of coefficient for each type of CPU jobs, which are listed in Table 4. The quality of this model is illustrated in Figure 17. The coefficients indicate that synchronization intensive CPU jobs such as CG are more sensitive to the bandwidth impact.

Table 4: Regression Analysis Results

Job Type	CG	EP	FT	IS	MG
Coeff. a	0.00761	0.0001	0.00563	0.00420	0.00457
Coeff. b	0.00151	0.007	0.00230	0.00641	0.00330

Although the coefficients varies across different types of CPU workload, a profiling-based threshold over cudaMemcpy bandwidth can be set up in production system to conservatively prevent cases of serious slowdown. In our experiment, for example, slowdown can be capped within 10% by preventing GPU jobs with bandwidth above 300MB/s from co-locating with any CPU job. In the following research, we will investigate the possibility of implementing a smarter threshold with estimation of the job-specific coefficients.

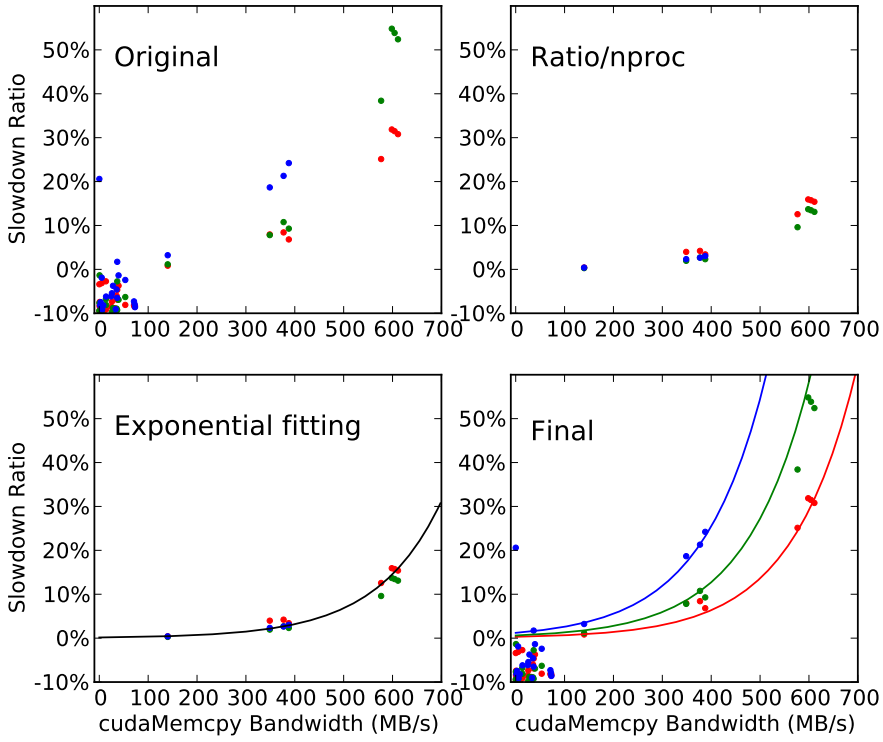

Figure 16: Regression Analysis of Slowdown over cudaMemcpy Bandwidth

5.1.4.4 Multi-node Jobs

When job collocation involves more than one node, the performance will be affected by other conditions such as network bandwidth. In this section, we scaled up the experiments to 4 nodes. The CPU jobs are configured with 32 processes ($ppn = 8$), and the GPU jobs are configured with 48 processes ($ppn = 12$).

Table 5: Average Slowdown of Co-locating Multi-node CPU Jobs with Triad GPU Jobs

Job Type	CG	EP	FT	IS	MG
CPU Job Slowdown	2.16%	15.1%	3.88%	4.89%	11.3%

Table 5 lists the CPU job slowdown of the extreme condition —- co-locating with Triad GPU jobs which incur the largest cudaMemcpy bandwidth and also a combined CPU load higher than 12. Even under such intensive pressure, the multi-node CPU jobs exhibit only a slight degradation in performance. For those communication

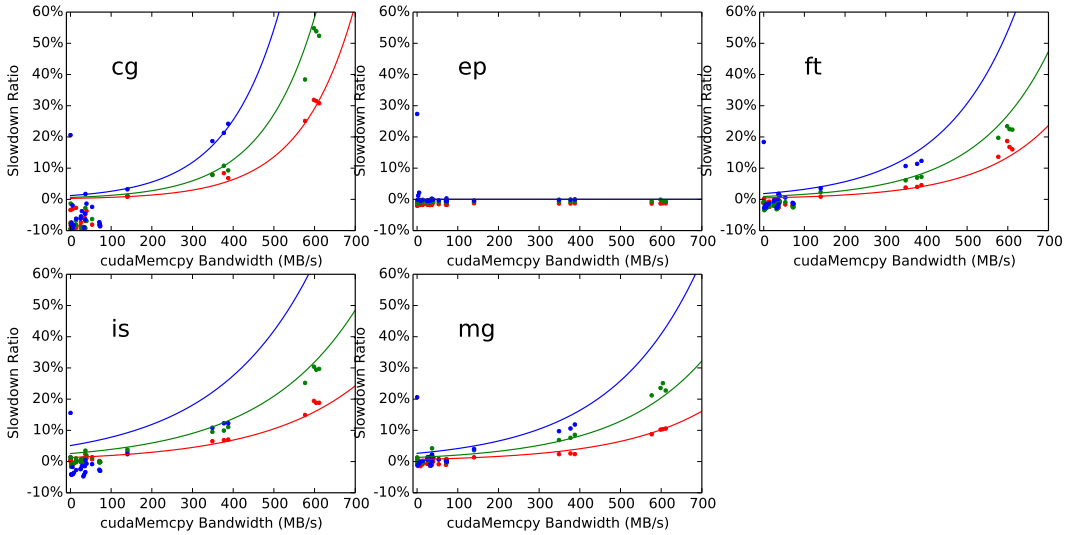

Figure 17: Regression Quality

intensive CPU jobs (CG FT and IS), the impact of job collocation is mostly shadowed by the impact of network due to the fact that the network bandwidth is significantly lower than the memory bandwidth. For those jobs with less communication (EP and MG), the slowdown ratios are relatively higher since the over-utilized CPU becomes the major bottleneck in those jobs. For the GPU jobs, the performance is identical to their single-node counterparts, because they are embarrassingly parallel on the process level.

5.2 Collocation-enabled System

In this section, an experimental system with collocation-aware job scheduler/resource manager is created to demonstrate the capability of job collocation in improving system utilization and throughput.

5.2.1 Insights on the performance of job collocation

The design of this collocation-enabled system is based on the insights gained from the studies of performance degradation. Our experimental results indicate that the

performance of job collocation can be determined by four major factors: 1) resource isolation, 2) CPU load, 3) GPU memory demands, and 4) inter-process synchronization/communication intensity. Among these factors, resource isolation can be easily managed by the system, CPU load and cudaMemcpy bandwidth can either be monitored by the system or be hinted by the user, and synchronization/communication intensity can only be hinted by the user. With these major factors informed, CPU/GPU jobs can be efficiently co-located to better utilize the heterogeneous resources.

5.2.2 Job Scheduling and Resource Management

In the new system, the jobs are scheduled according to their priority. GPU jobs are assigned higher priority than CPU jobs since the system can be relatively better utilized by GPU jobs. In either type of jobs, the ones with higher *ppn* value have higher priority. Backfilling is also enabled, so that some of the low priority jobs (most likely the small CPU jobs) may be started earlier if they are not delaying high priority jobs. The objective of this scheduling policy is to optimize the overall system utilization.

The CPU resources are managed in an elastic way. The CPU jobs are given 100% dedicated CPU resources during their requested execution time, but the GPU jobs are given 100% dedicated CPU resources only during the monitoring period of their requested execution time. After that, the underutilized CPU resources may be collected if there are feasible CPU jobs. Specifically, when a GPU job gets executed, the system begins monitoring the CPU load of its allocated nodes at 1-minute resolution. The monitoring period ends when the job reaches portion P_M of its requested execution time, and then an estimation of the underutilized CPU resources is made based on the CPU load assuming that the job's future behavior will be consistent with the behavior we monitored. The underutilized CPU will be marked as available until the GPU job finished, allowing feasible CPU jobs to be filled into these temporal

resources.

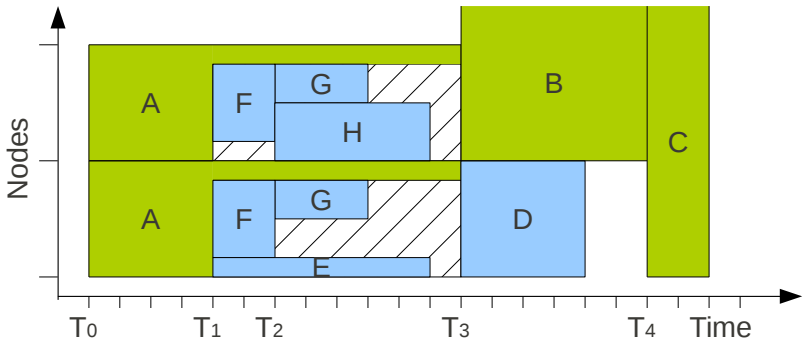

Figure 18: Scheduling Example of Mixed Jobs

Figure 18 gives a scheduling example. Job *A*-*C* are GPU jobs, and job *D*-*H* are CPU jobs. The portion of monitoring period is set as $P_M = 1/3$. At the beginning, job *A*-*D* were scheduled to time T_0 T_3 and T_4 based on their priorities. At time T_1, the system finished monitoring job *A* and also marked the underutilized CPU resources available on the related nodes. Feasible small CPU jobs, *E*-*H*, were then scheduled to time T_1 and T_2 on the temporary resources.

Since the estimation of temporary underutilized resources may be inaccurate due to irregular behavior of some GPU jobs, an upper bond P_B can be set on the maximum collectible CPU resources. For example, with $P_B = 8$, at most 8 CPU processes are allowed to be scheduled on the temporary resources, even if the monitored CPU load is only 1.5 for the existing GPU job. Such limit can create a buffer zone for the GPU job to accommodate unpredicted contention.

The combination of P_M and P_B determines the resource collection policy. A moderate policy of longer monitoring period and smaller collectible CPU resources limit can lower the chance of jobs being killed due to serious delay.

5.2.3 Implementation Details

In order to support the new feature of job collocation, modifications are needed in Torque [117] and Maui [67]. Based on the idea that job collocation is more of

a scheduler feature than a resource manager feature, we implemented the most of the new functionalities in Maui, while only keeping resource monitoring and binding duties in Torque.

5.2.3.1 Torque

Torque is a very popular open-source HPC resource manager based on the original PBS project. Torque has three components: a) pbs_server, a management deamon running on central node; b) pbs_sched, a basic scheduler deamon running on central node; and c) pbs_mom, a monitor deamon running on each computation node. Instead of relying on its basic scheduler, Torque is usually deployed with another standalone scheduler such as Maui to achieve better performance.

The major modification we made on Torque 2.5.12 is recording PID of MPI processes and manipulating CPU binding. Although binding is natively supported in Torque code, the affinity can not be altered dynamically. Therefore, we modified the its scheduler API to record and report the MPI processes' PID of each job. Maui will make the binding decision and inform Torque's pbs_mom to manipulate the binding dynamically. To better monitor the CPU load, we also inserted a time stamp on each CPU load value captured.

5.2.3.2 Maui

Maui is a widely used open-source batch job scheduler, and it supports a variety of scheduling features. However, it has not been optimized for GPU cluster. We have implemented three new features on top of Maui 3.3.1: a) GPU-aware node allocation, b) Elastic CPU resource management, c) Collocation-aware dynamic CPU set binding.

With these new features, the scheduler can prioritize qsub request with specific GPU requirement. It will monitor and collect underutilized CPU resource on those nodes occupying by GPU jobs. When co-located jobs are detected, the scheduler is

also capable of binding them to partitioned CPU resources.

5.3 Experimental Result

In this experiment, the system is presented with a pool of mixed jobs, in which the 32 GPU jobs and 300 CPU jobs are implemented with the workloads that are studied in previous sections. The *ppn* is configured as 2-8 for the CPU jobs and 12 for the GPU jobs. The execution time is configured as 0.5-11 minutes for CPU jobs and 7-13 minutes for the GPU jobs. When submitting each job, the time request is set to 115% of its expected execution time. The system is configured with a moderate resource collection policy: the upper bond of collectible CPU resources P_B is set to 9 out of 12 cores, and the portion of monitor period $P_M$is set to 1/3.

When the jobs are handled by conventional Torque/Maui system, it takes a makespan of 9345 seconds to finish. In the entire 448,560 CPU core seconds, 195,162 are spent on CPU jobs, 237,936 are spent on GPU jobs, and 15,462 are idle due to scheduling. The weighted average CPU load for GPU jobs is 2.5, which means about 188,000 core seconds are underutilized in GPU jobs. The effective CPU utilization during entire makespan is 54.5%.

When handled by our collocation-enabled Torque/Maui system, the makespan is 7,857 seconds, reduced by 15.9%. The CPU time spent on either type of jobs remains roughly unchanged since the scheduler is aware of the potential slowdown when co-locating jobs. 8,118 idle core seconds are created by scheduler. The results indicates that the co-located CPU jobs have scavenged more than 60,000 or 32% underutilized core seconds from GPU jobs. The effective CPU utilization is improved to 64.0%, and the GPU utilization is also improved form 53.0% to 63.1%. In average, only 2.3 out of 332 job are killed due to timeout, and they are short CPU jobs.

In the 188,000 core seconds of underutilized CPU resources, 33.3% resides in the monitoring period, 10.3% is due to the bond of collectible resources, and the rest 56.4%

is the real collectible resources. If the moderate collection policy is strengthened to $P_M = 1/5P_B = 2$, the real collectible resources will increase to 75.1% even though the timeout cases are expected to increase.

Our results indicate that co-locating CPU job and GPU job is a feasible way of improving system utilization. With those factors properly informed and managed, the co-located CPU job can efficiently scavenge the underutilized resources without incurring significant performance degradation in either of the co-located jobs. Our experimental 4-node system achieved 15.9% gain in throughput and 10% gain in both CPU and GPU utilization with a moderate resource collection policy. With a stronger policy, the system is capable of collecting 75% of the underutilized CPU cycles.

5.4 Related Works

Our solution of co-locating CPU-only jobs with GPU-assisted jobs can be categorized as a specific mechanism to overlap CPU/GPU computation. In general, there are two levels of CPU/GPU computation overlap: intra-process and inter-process.

Overlapping CPU/GPU computation within a application process requires a tactful scheme for workload partitioning and pipelining. Although the GPU library (CUDA or OpenCL) does not preclude the process from forking multiple threads and running GPU kernels asynchronously, to design such a platform-dependent and workload-dependent scheme can be very challenging for the developers. Extraordinary programming and tunning efforts are devoted into overlapping CPU/GPU computation in specific applications, such as linear algebra [43, 126], stencil computing [124], seismology [100], and bio-medical imaging [122]. A number of recently proposed systems [69, 87, 37, 103] attempt to automatically decompose and schedule the computations on CPU and GPU, but most of them are relying on more restrictive programming model and/or extensive training to construct a well overlapped computation scheme.

Overlapping CPU/GPU computation among multiple jobs is implicitly supported by the operation system. Due to the possible contention created by co-located jobs, such level of overlap should be facilitated by specially designed software infrastructures to avoid performance degradation. Previous research [64] showed that throughput of traditional CPU-only clusters could be improved by co-locating jobs to oversubscribe CPU resources. The topic of idle CPU resource scavenging has also been studied in the context of grid computing [11] and volunteer computing [119] (e.g. the SETI@home project [12]). To the best of our knowledge, our work is the first systematic study of co-locating CPU-only jobs with GPU-assisted jobs to improve the utilization and throughput of a GPU cluster.

CHAPTER VI

HADOOP CLUSTER OVERVIEW

With enormous amount of data created every second from Internet services, research labs, shops and factories, as well as our everyday life [86], the challenges of data storage and processing are greater than ever before. Extraordinary efforts have been devoted into the development of new computation infrastructures that can handle the exponentially growing data cost-effectively. Among the technologies created for data-intensive applications, the open-source project Hadoop [54] has attracted major attention.

Hadoop is a cluster computing system designed to store and process an extensive amount of data with high throughput. The development of Hadoop is inspired by Google's proprietary MapReduce [48, 34] data processing system. Rather than relying on highly-engineered hardware, the Hadoop system can reliably scale out to thousands of low-cost commodity computers. The design ditches those expensive components of traditional HPC clusters, such as the dedicated storage system and the proprietary high-performance network, and consolidates computation and storage onto a simple Ethernet-based multi-rack cluster infrastructure. The Hadoop system incorporates a series of software techniques, including locality-aware scheduling and automatic data replication, to compensate for the less reliable hardware and the low-bandwidth network.

With the rapid advancement of the open-source project and the diversifying demands of data-intensive applications, the Hadoop clusters are becoming the foundation of a thriving big-data ecosystem. Popular members of this ecosystem include

data-mining engine Mahout [97], in-memory processing engine Spark [142], graph processing engine Giraph [14], data warehouse tools Pig [95] and Hive [123], and tabular storage HBase [47]. Since these technologies are either entirely developed on Hadoop or relying on HDFS for storage, the performance of Hadoop is critical to the whole ecosystem.

The objective of this research is to improve the throughput of a Hadoop cluster with congested network resource. Specifically, we are focusing on improving the efficiency of remote I/O traffic by leveraging the parallelism inside the source and destination of the data flows. In this chapter, we first give the overview of Hadoop cluster and its limitations in remote I/O. Our solutions of multi-source streaming and multicast-based replication are then described. The related researches are also discussed in this chapter.

6.1 System Overview

Hadoop is developed with the abstraction of a 3-level hierarchical cluster structure that resembles the arrangement of common computing clusters and data centers. The computer nodes in a Hadoop cluster are evenly packed into racks and connected to the Ethernet switch on top of each rack. The rack switches are then connected to a core switch. The in-rack bandwidth is usually much larger than the off-rack bandwidth.

The fundamental design concept of Hadoop is moving computation to data, because in many data-intensive applications the cost of moving big data sets can actually outweigh the cost of processing them. Executing the computation task on a node that contains the input data is a primary scheduling objective. When data-local execution is infeasible, executing in-rack is still preferred to executing off-rack. The pursuit of such data locality not only accelerates the applications, but also saves the precious bandwidth for the Hadoop cluster.

The two major software components in a Hadoop cluster are Hadoop MapReduce and Hadoop Distributed File System (HDFS). Hadoop MapReduce is a batch processing system for large-scale data-intensive workload. It uses a centralized job manager (the JobTracker) to handle job submission and scheduling, and uses a fleet of distributed task managers (the TaskTrackers) to execute the individual tasks on the computer nodes. HDFS is a distributed storage infrastructure designed for MapReduce applications. HDFS supports the write-once-read-many semantics and is optimized mainly for the streaming data access pattern. The design of HDFS is based on several important observations in solving big data problems: 1) Hardware failure is a norm rather than an exception; 2) The size of individual data set is usually gigabytes or terabytes; 3) Data sets are modified only by appending new entries rather than overwriting existing entries. Similar to the processing system, HDFS also consists of two parts: a centralized NameNode and a fleet of distributed DataNodes. The NameNode stores all the metadata and exposes a unified namespace to the applications. When the application (e.g. a MapReduce task) initials a read or write request, the NameNode will locate a group of DataNodes to serve the request. Each individual file is stored in HDFS as a sequence of fixed-size blocks. The blocks are replicated across DataNodes for availability and performance purposes. The block size and replication factor can be configured on per file basis. Typically, the file is store as 64MB-256MB blocks with three replicas for each block.

The TaskTracker and the DataNode are co-located on each node of the Hadoop cluster. In most cases, the logical input split of a MapReduce task can be retrieved from a physical data block on HDFS. Such one-task-one-block mapping is the key for a Hadoop cluster to exploit data locality in the applications.

6.2 Limitations of Native Hadoop System

Due to the large volume of data-intensive applications and the limited network resources available in commodity clusters, the remote I/O throughput is one of the major bottlenecks for such system. Although the Hadoop system is developed to move computation to data, remote data read and write between MapReduce jobs and HDFS DataNodes are still inevitable. The limitations existed in the native implementation of Hadoop could potentially compromise the throughput of the cluster in many cases.

6.2.1 Static Single-sourced Remote Data Access

Exploiting input data locality is a very important design of Hadoop, however such locality is not always attainable in practice. Previous study about MapReduce workload in Facebook and Yahoo clusters [141] revealed that, if the default FIFO scheduling policy is applied, the small-sized jobs (which represents more than 80% of total jobs studied) are likely to read their input from remote sources, because the policy will force such jobs with no local data to be scheduled. Data locality is also hard to obtain for the dual-input jobs. Many popular MapReduce applications, including the PageRank for machine learning and the join operation for data warehousing, actually require more than one data block per task. In these jobs, it is very unlikely that a task can find all the input data on local node. Another case that compromises locality is cloud-based cluster. Nowadays, with the help of cloud services such as Amazon EC2, large-scale computation can be handled in a cost-effective way which is vital to small businesses. However, building Hadoop clusters over the cloud can void the rack-awareness of HDFS and MapReduce, because cloud service providers usually give little or no guarantee on the network topology of their virtualized computer instances. Moreover, on some virtualized instances (e.g. Amazon Elastic Block Storage based) even the "local disks" are implemented with network storage.

In the native Hadoop implementation, the map task reads input with the file system client (DFSClient). Before reading a data block, The DFSClient first contacts the NameNode to locate the block. If the block is not locally available, the client will retrieve the data with a TCP-based BlockReader. The client will always try to stream the data from the nearest DataNode that contains one of the replicas. The awareness of network distance (in number of hops) is based on a pre-defined network topology. If such information is unavailable, the system will treat all remote nodes as one hop away.

Streaming remote data with such static single-source mechanism has several limitations. First of all, the source is selected solely by the static topology. The nearest source is not necessarily the fastest one, especially when that specific network path is suffering from congestion. Without the awareness of resource contention, the mechanism is also unable to choose the correct candidate if multiple sources have the same distance. Moreover, even if the fastest source is selected, streaming from that single node may not be able to fully utilize the available network bandwidth.

6.2.2 Inefficient Data Replication

Unlike the read operation, which can be done either locally or remotely, a write operation to HDFS always sends data over the network because HDFS replicates each piece of data onto multiple nodes across the cluster to guarantee data availability and to boost accumulated read throughput. According to previous studies [27, 80], the replication process can account for up to half of the total network traffic in production Hadoop clusters.

In native Hadoop implementation, replication is carried out in a pipelined procedure. Before the writing a new block, the DFSClient contacts the NameNode to get a list of DataNodes as the destination of the replicas. Suppose the replication factor is three. The client only writes to the first DataNode on the list, and the data is sent

as a sequence of small packets (64KB each by default). The packets are immediately relayed from the first DataNode to the second, and from the second to the third.

The relaying data flows created by pipelined replication mechanism are potentially redundant, since the same pieces of data are transferred back and forth between the DataNodes and the switching fabric. Due to the large volume of replication traffic, such redundancy can waste a significant amount of the network resource and adversely impact the throughput of entire Hadoop cluster.

6.3 Proposed Solutions

6.3.1 Multi-source Streaming

In this research, we are first focusing on improving the remote read throughput by slicing the file block and streaming the slices from multiple sources, so that all available replicas can be utilized simultaneously to boost throughput. To enable this feature in Hadoop, a SliceReader is implemented to replace the native BlockReader, and a new circular buffer is introduced in DFSInputStream to coordinate the multiple readers.

The advantages of the multi-source streaming include (1) the potential of achieving higher throughput by utilizing the bandwidth of multiple sources; (2) the independence from relying on predefined network topology to explore locality; and (3) the ability of adapting to dynamic congestions on network as well as on other resources.

Experiments are conducted on several hardware platforms to evaluate the feasibility and the performance of the new system. The results show that multi-source streaming can effectively improve the remote read performance of a Hadoop cluster. On the 32-node Amazon EC2-based system, the TestDFSIO benchmark is accelerated by as much as 20%, the TeraValidate benchmark is accelerated by 15%.

6.3.2 Multicast-based Replication

The second part of this research is focused on improving the write throughput with an multicast-based replication procedure. By utilizing the multicast feature of network switches, our design can significantly reduce the amount of traffic needed for replication. Figure 19 shows an example of writing three replicas over the network. In this scenario, pipelined replication creates three north-bound (from node to switch) data flows, whereas multicast-based replication creates only one such data flow. By reducing the redundant flows, the multicast-based method can save bandwidth for other concurrent network traffic, and also improve the write throughput of the jobs, since the data flow is now going through less number of network paths and becomes less vulnerable to congestion.

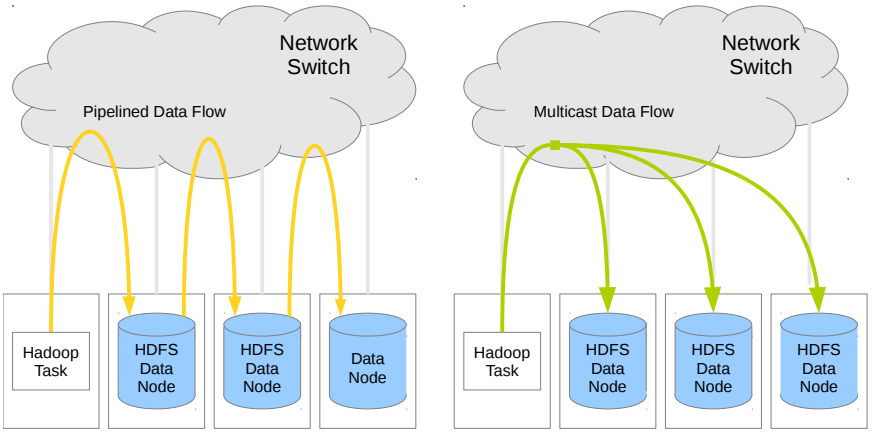

Figure 19: Pipelined Replication VS Multicast-based Replication

The backbone of our system is a Java-based fast reliable and congestion-controlled multicast socket implementation named CCRMSocket. This multicast socket is specifically developed for Hadoop cluster to replace the pipelined TCP sockets. The interface between CCRMSocket and HDFS are carefully designed to guarantee the backward compatibility of the system. The performance of CCRMSocket is tested with our multi-rack platform. The experimental result shows that the CCRMSocket can effectively save network bandwidth and peacefully coexist with other TCP traffic

flows.

We also developed a flow-based Hadoop simulator (HFlowSim) to study the impact of replication traffic to the performance of a large-scale Hadoop cluster. This simulator can take real Hadoop job trace as input and simulates the network traffic created by the jobs. The simulation result suggests that multicast-based replication can improve a Hadoop system by the accelerating the big jobs that have large amount of data to read and/or write.

6.4 Related Works

Since network bandwidth is critical to data-intensive applications, considerable efforts have been devoted to improve data locality by seeking better placement of workload and data. Multiple studies are also conducted to directly improve the network performance in a Hadoop cluster or in a data center scenario.

6.4.1 Workload Placement

Hadoop's optional fair schedulers and capacity scheduler were designed to solve the small job starvation problem in the default FIFO scheduler. Delay Scheduling [141] proposed a wait technique to improve the locality: when no task from a job can be assigned to a local node, the scheduler waits for several turns such that more nodes will be available to schedule the job. Quincy [65] transformed the scheduling problem into a minimum cost flow problem which took fair constraint into account as well. The graph vertices represented tasks, nodes and racks. Each edge was a possible task assignment on a node or a rack. Edge cost and capacities represented the locality cost and the fairness constraints.

LARTS [57] and CoGRS [56] discussed optimal reduce task placement. The placement of reduce tasks was obtained by choosing the node closest to all the map tasks in a rack that hosts the maximum number of map tasks. The data size skewness of map tasks was also considered in CoGRS to alleviate the reduce straggler problem.

6.4.2 Data Placement

DiskReduce [41] applies the design of RAID devices into HDFS by proactively erasure-coding any data block after 24 hours from its birth. The motivation behind such uniform dereplication is the temporal accessing locality in Hadoop clusters that most of the accesses take place during the first day of file creation. This technique is further improved in the work of Ursa Minor [2] and ERMS [28], where the storage scheme (in terms of replication number or erasure-coding plan) of each file can be adjust dynamically. However, the adjustment process is not spontaneous. It depends on the specifications given by the user.

Scarlett [9] uses an off-line method to adjust the replication number of popular files. Its heuristic minimizes the total number of hotspots in the cluster with a limited space budget. DARE [1] attacks the same problem with a cache-like on-line approach. It creates dedicated space on each DataNode to store dynamically created replicas. The caching and eviction are performed based-on a randomized heuristic. CDRM [130] also incorporates dynamic replication in the design, and the main objective of this system is to optimize the data availability.

Purlieus [99] discussed the data placement in terms of locality for both map and reduce stages. The authors proposed to couple the scheduling decision on both data placement and task placement to reduce the amount of data transfer in a cloud environment. Applications were categorised into map-input heavy, map-and-reduce-input heavy and reduce-input heavy jobs. Placement techniques were proposed for each of the job type. PACMan [10] discussed the memory locality problem. A major discovery in their work was that hit-ratios do not necessarily improve job completion times and therefore special coordinated cache replacement policies were developed.

6.4.3 Network Improvements

A network levitated merge mechanism was developed in [129] to exploit RDMA in a manner that the data does not have to be copied to disk. They have also developed a shuffle-merge-reduce pipeline that works in conjunction with the RDMA-based merge. Another pair of projects have also used Infiniband and RDMA to improve the performance of Hadoop HDFS [66] and HBase [63].

Several other techniques have been presented to enhance data center networks, for example, by enabling multipath [106], flow priority [61], and bandwidth control [104]. Since the implementation of adaptive multi-source streaming and multicast-based replication are compatible to most of those techniques, a versatile enhancement solution of network utilization can be orchestrated [30] to further benefit Hadoop cluster and the ecosystem on top of it.

CHAPTER VII

MULTI-SOURCE STREAMING ON HADOOP CLUSTER

In this chapter we are going to study the possibility of improving the remote read performance of Hadoop cluster by streaming input data from multiple replicas. We first present a novel multi-source streaming enabled Hadoop system, which incorporates a new circular buffer, a new slice reader, and the enhanced DataNode with new slice transfer protocols. Detailed benchmark and analysis are then conducted to support the design of this system. The new system is verified with multiple experiments, and the result shows significant improvements over native Hadoop system in the overall throughput and also the robustness to imbalanced system congestion.

7.1 System Framework

In Hadoop applications, the input data is typically accessed in chunks with a buffered streaming reader. For example, text-based data is often accessed in $64KB$ chunks with the LineReader, whereas binary data can be accessed in any predefined chunk size with the SequenceFileReader. In a native Hadoop task, which is implemented with single thread, the input data chunks are sequentially read and processed, as illustrated in Figure 20. Such implementation prohibits the task from utilizing multiple data replicas to boost its throughput.

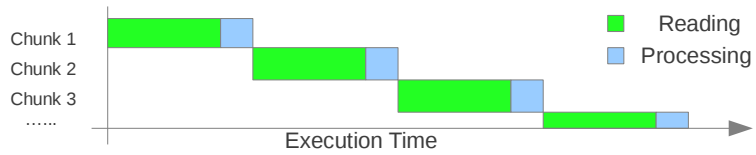

Figure 20: Task Execution Pattern in Native Hadoop System

The basic idea of our design is to pipeline the execution into separate producer (the

reader thread) and consumer (the processor thread). Reader and processor are able to work asynchronously with an extra buffer layer implemented between them. Multiple readers can be launched to read different subsets of the data from multiple replicas at the same time. The data will be re-ordered in the buffer and then supplied to the processor. The overall throughput is expected to be improved over reading all data from single source. Because all the readers are fetching data with their best effort, an under performed reader (who suffers from greater disk or network congestion) is likely to lose the competition with its peers and thus contribute less data into the buffer. Such competition mechanism adaptively guarantees the remote data to be accessed at high throughput.

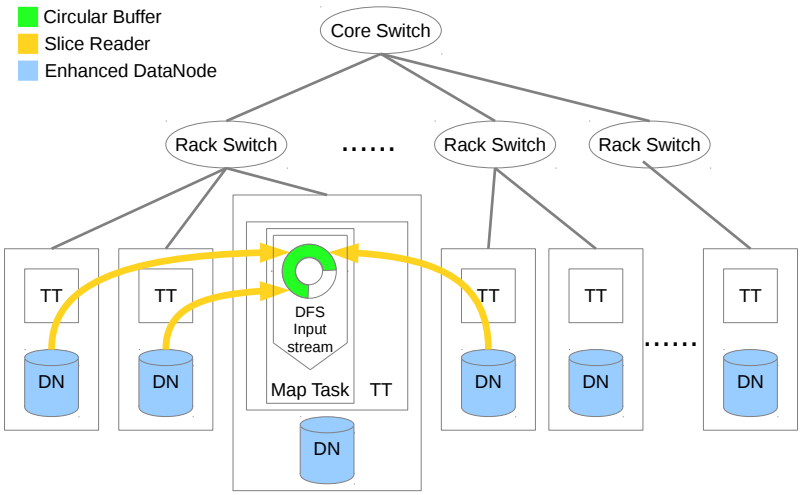

Figure 21: Framework of Multi-Source Streaming Enabled Hadoop Cluster

Figure 21 shows the framework of our multi-source streaming enabled Hadoop cluster, in which the three major components are marked with colors.

1. *Circular Buffer* Instead of reading with native BlockReader, our DFSInputstream is reading the data split from the new circular buffer. The introduction of this buffer breaks the synchronized binding between map task and reader, thus enables the data to be asynchronously produced and consumed. The data is buffered at the granularity of slice. All data slices are of the same size, which

by default is much smaller than the size of the buffer. Once enough data is consumed by the task (either being read or skipped), new data slice would be fetched into the buffer. In order to cope with the bandwidth variation among the readers and to smoothly stream the buffered slices to the task, a buffer manager is implemented to support advanced slice assignment policy.

2. *SliceReader* The slice reader in our system is an extension of Hadoop BlockReader with multi-threading capability. Similar to the native design, each reader is bound to the DataNode server of specific source during instantiation. However, the new slice reader is optimized for reading disjointed data slices rather than a whole consecutive block. Once the reader receives a slice assignment from the buffer manager, it will start fetching data into associated space of the buffer with its best effort. When this slice is finished, the reader will wait for another assignment while retaining the network connection. The reader can also be interrupted, if the manage wants to withdraw current assignment or to shut-down this reader.

3. *Enhanced DataNode* With native Hadoop system, there are two methods to read two disconnected subsections of a file block. One is invoking single block reader to read from the beginning of the first subsection to the end of the second subsection and discard any irrelevant data between the two. The other is invoking two block readers each reading one subsection, but this method doubles the overhead of setting up network socket. In order to reduce the overhead, we implemented a new service protocol in DataNode server coded as OP_READ_SLICE) allowing disconnected slices to be transferred via a persistent socket connection. With this enhancement, slice reader can issue new request with existing I/O stream once previous read finished. Although each request still costs at least 1 RTT (network round-trip-time), it's much more

efficient than using two native methods.

7.2 Challenges and Solutions

In this section, we first introduce the major challenges involved in the design of our multi-source streaming system and then present the detailed design solutions based on experimental evaluation and theoretical analysis.

7.2.1 Design Challenges

7.2.1.1 Slice Size

Intuitively, streaming one block of data from multiple sources may have both positive and negative impacts over the performance of map task. On the positive side, the multi-source streaming enabled system will benefit from higher aggregated reading bandwidth and overlapped processing. On the negative side, since our system need multiple small transactions rather than one large transaction to read one remote block, overhead may incur from scattered disk access and extra network RTTs.

According to our slice-based design, the remote reading overhead is affected by two aspects: the size of slice and the assignment of slices. The slice size determines the granularity of the streaming system. Larger slice size can lower overhead by increasing the efficiency of disk access and reducing the total number of network transactions. However, smaller slice size can reduce the total buffer size and also improve the system's responsiveness in adaptation to bandwidth variations.

The assignment of slices determines the scatterness of disk access. In our multi-source streaming system, multiple readers are aggressively competing for slices. Each reader will eventually acquire a subset of the total slices proportional to throughput its source can deliver. For example, if there are two remote sources with identical capability, then each of the two readers will read about 50% of the slices. Moreover, since the slices are assigned in order, the slice subset acquired by one reader will always spread across the whole block. As a result, a smaller subset will lead to

relatively higher scatterness. A reader with a subset of percentage r will only access one slice in roughly every $1/r$ consecutive slices.

In our systems, only the slice size is explicitly defined. The relative size of each reader's slice subset is the result of their competition, so that the system is capable of adapting to performance variation. The foremost challenge in designing the system is to find an appropriate slice size.

7.2.1.2 Streaming Order

In multi-source streaming enabled Hadoop system, another important aspect that determines the execution performance of map task is the arriving order of data slices, because the slices have to be processed in-order. In native system, such arriving order is implicitly preserved, since the entire block is streamed from single source. However, in the new system, slices may arrive out-of-order due to throughput differences of parallel readers. Although the MapReduce model guarantees that the input of splittable key-value pairs can be processed in arbitrary order, the raw data has to be processed sequentially since the storage system API (DFSInputstream) is agnostic to the boundary of user defined data entries. Given arbitrary slice, the record reader may not be able to retrieve the data entries in it, because the related meta-data could reside in another slice.

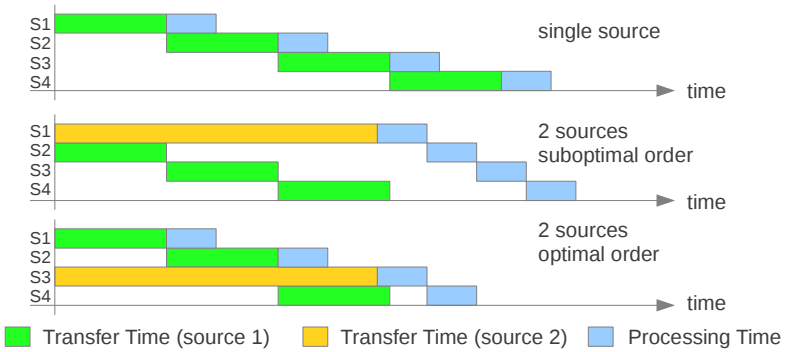

Figure 22: Task Execution Flow with 4 Slices and 2 Sources

Due to the requirement of in-order processing, blindly fetching data from multiple

sources doesn't guarantee a minimal execution time. As shown in the example of Figure 22, simultaneous fetching 4 slices from 2 sources may actually extend the execution time if the slices are transferred in a suboptimal order. So the second design challenge is to find out and to maintain a good streaming order of the slices.

7.2.2 Optimizing the Slice Size

7.2.2.1 Performance Benchmarking

We developed a micro benchmark to evaluate the remote read performance over scattered data slices. In this benchmark, a client continuously reads a subset of data slices from a file hosted on the remote server. By specifying the slice size and subset percentage parameters, the benchmark can emulate the reading pattern of a map task in the multi-source system. For example, with the parameters set to $1MB$ and 33%, the data will be divided into $1MB$ slices and the client will access about one in every three slices. The benchmark will terminate once $256MB$ of data has been fetched by the client. The size of the entire file is set to a very large value so that the interference of Linux page cache can be avoided. This micro benchmark is implemented with similar functions and protocols as Hadoop does (except for setting checksum off).

Four platforms, which represent several different types of Hadoop clusters, are used in this evaluation. Cluster A is a private cluster connected with a Gigabit Ethernet switch. Cluster B is a KVM-based virtual cluster hosted inside a multi-core machine. The nodes are connected with OpenvSwitch and network bandwidth is configured to $1Gbps$. To minimize interference, dedicated CPU RAM and HDD resources are assigned to each node. Cluster C and D are cloud-based virtual clusters. Both clusters use EC2 medium instances as their nodes and have identical network performance, but the disk storage is configured differently in the two clusters. More detailed specifications can be found in Section 7.4 Table 6.

As shown in Figure 23, the remote reading throughput improves with increased slice size and subset percentage. The throughput recorded by the benchmark covers

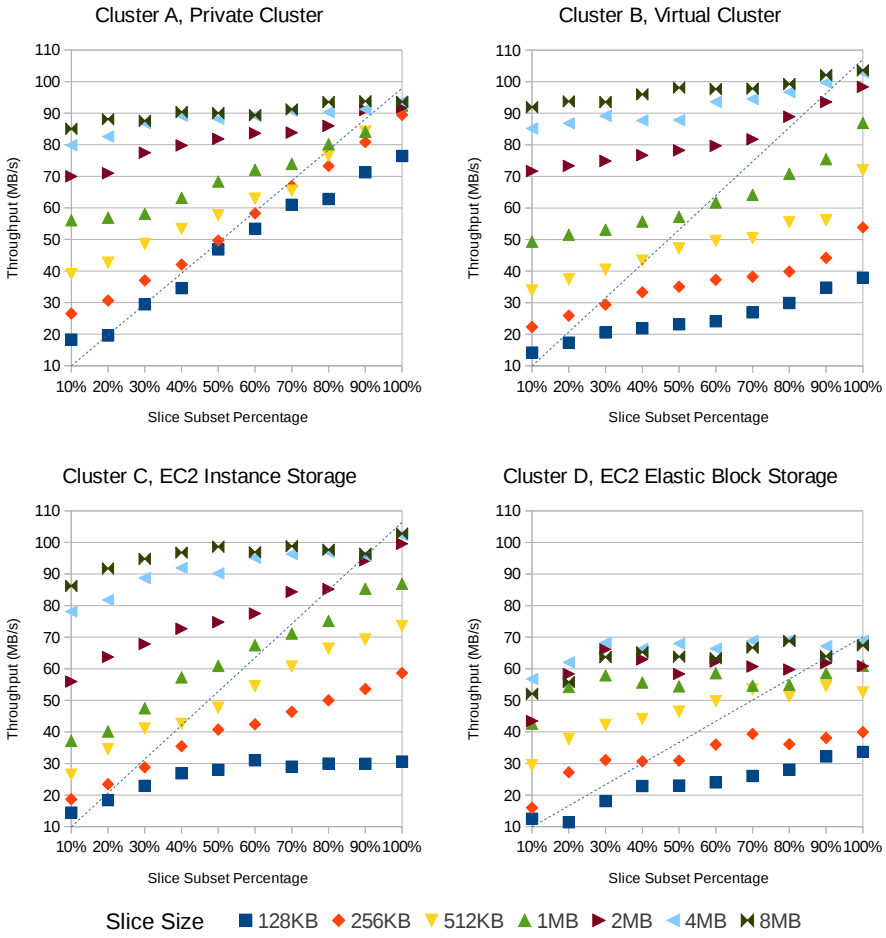

Figure 23: Benchmark Result of Remote Data Accessing Throughput

both disk performance and network performance. For Cluster A-C, the throughput is capped at about $100MB/s$, which is limited by the $1Gbps$ in-bound connection bandwidth. For the Amazon EBS-based Cluster D, the throughput is capped at about $70MB/s$ by the EBS disk performance.

7.2.2.2 Slice Size Selection Heuristic

In order to choose an appropriate slice size, we use a diagonal line as reference. The diagonal line illustrated in the Figure 23 can be used as a heuristic indicating whether specific slice size is large enough to benefit multi-source streaming system. For a specific system platform, if the all the throughput records of certain slice size are

located above the reference line, streaming from multiple sources should yield a higher aggregated throughput than the native single source method. Take Cluster C as an example, if the data are streamed from 3 sources with $2MB$ slice, the aggregated throughput of readers with any subset distribution will be larger than $100MB/s$. However, if the data are streamed in $256KB$ slices, the aggregated throughput is only about $90MB/s$ when the subset distribution is 50%-30%-20%.

This heuristic can be formulated as follows. For a specific system platform, if native single source streaming can yield a throughput of $C\ MB/s$ (which is marked with a star in the figure), the diagonal line represents the throughput function $T(r) = Cr$, in which r is the subset percentage. Suppose the slice-based streaming throughput is $T'(x, r)$, a function of slice size x and subset percentage r. Then the aggregate throughput of streaming from multiple source can be derived as $\sum_i T'(x, r_i)$ with $\sum_i r_i = 1$. A sufficient condition that makes multi-source streaming outperform the native method is $T'(x, r_i) > T(r_i)$ for each source i, because we will have $\sum_i T'(x, r_i) > \sum_i T(r_i) = \sum_i Cr_i = C$ under such condition.

According to the benchmark result on our experimental clusters, we decided to set the slice size to at $2MB$ in our multi-source streaming system. Since streaming at $2MB$ slice size has already exhibited a $100MB/s+$ throughput which is close to the practical bandwidth limit of Gigabit Ethernet, we expect this slice size to be feasible for other Hadoop clusters implemented with such hardware. With fixed slice size, the total size of the buffer determines how many slices it can maintain concurrently. A larger capacity is necessary if the throughput of multiple readers vary significantly, otherwise the slow reader is unable to cooperate with faster reader. In practice, we choose to maintain a $16MB$ buffer, which is able to host up to 8 slices and accommodate a $7\times$ throughput difference across the readers. An advanced interruption mechanism will also be introduced in following subsections to prevent the task from being blocked by the slowest reader.

7.2.3 Optimizing the Streaming Order

7.2.3.1 Modeling Analysis

In Hadoop system, the streaming problem can be modeled as transferring a block of p file slices $\{s1, s_2, ..., s_p\}$, each containing D bytes of data, from a group of k source nodes $\{n_1, n_2, ..., n_k\}$ to one single map task. A slice may be transferred from any source and may arrive at any time, so we use $\{b_1, b_2, ..., b_p\}$ to define a specific arriving order, in which b_i is the time when slice s_i is completely fetched into the buffer.

The objective of this streaming problem is to minimize the map task's execution time. We model map task's execution flow of our multi-source streaming enabled system as consecutively reading and processing each slice until all the p slices are finished. We further assume that processing each slice costs a constant time c due to the identical slice size. So, based on our reading-processing overlapped execution design, the completion time t_i of slice s_i can be expressed as

$$t_i = \begin{cases} b_i + c & \text{if } i = 1 \\ \max(t_{i-1}, b_i) + c & \text{otherwise,} \end{cases} \tag{1}$$

and the execution time t_{map} can be computed as

$$\begin{aligned} t_{map} = t_p &= \max(t_{p-1}, b_p) + c \\ &= \max(\max(t_{p-2}, b_{p-1}) + c, b_p) + c \\ &= \max(t_{p-2} + 2c, b_{p-1} + 2c, b_p + c) \\ &= \max_{i \in [1,p]} \{b_i + (p + 1 - i)c\}. \end{aligned} \tag{2}$$

t_{map} can be minimized in two steps. The first step is to minimize $\{b_1, b_2, ..., b_p\}$ by saturating the network bandwidth. This step can be achieved by implementing an aggressive reading mechanism. Whenever a source becomes idle (finished a previous slice), its associated reader will immediately request a new slice from the buffer manager and start fetching data into the buffer. According to the previous subsection, the in-bound network bandwidth should be saturated with a large enough slice size.

The second step of minimizing t_{map} is to optimize the order. We are going to prove by contradiction that t_{map} will be minimized if the slice arriving order $\{B_1, B_2, ..., B_p\}$ satisfies the constraint

$$B_i \leq B_{i+1}, \forall i \in [1, p). \tag{3}$$

Without loss of generality, we assume

$$t_{map}\{B\} = B_x + (p + 1 - x)c. \tag{4}$$

Suppose there exists a permutation $\{b_1, b_2, ..., b_p\}$ of $\{B_1, B_2, ..., B_p\}$, which makes

$$B_x + (p + 1 - x)c > b_i + (p + 1 - i)c, \forall i \in [1, p]. \tag{5}$$

Since c is a positive value, we will have

$$B_x - b_i > (p + 1 - i)c - (p + 1 - x)c \geq 0, \forall i \in [1, x]. \tag{6}$$

Inequality (6) guarantees that there are x elements strictly smaller than B_x. However, it is contradictory to inequality (3) that B_x can be larger than at most $x-1$ elements.

7.2.3.2 The Adaptive Streaming Heuristic

According to above studies, the multi-source streaming system should be designed with two goals: saturating the in-bound bandwidth and preserving the slice arriving order. In an idealized environment, where the out-bound bandwidth of each source is stationary and independent, an optimized streaming order can be easily constructed. In practical environment, however, such goals can only be approached by adaptive heuristics.

In our streaming design, each DFSInputstream has a fixed-size circular buffer and a buffer manager. The manager maintains a list of IDs representing all the buffered slices (already in the buffer) and scheduled slices (to be fetched into the buffer). The head ID will be removed when that slice is consumed, and a new ID will be append to

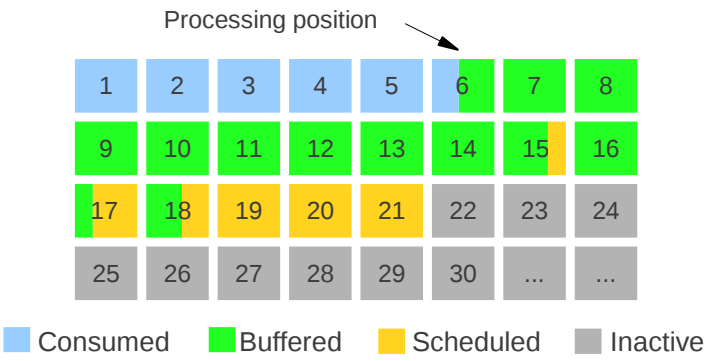

Figure 24: A Buffering Example with Capacity of 16 Slices

the list if there is inactive slice (not yet scheduled) available in the block. Figure 24 gives a buffering example with a buffer capacity of 16 slices. In this example, three slice readers are working on s_{15} s_{17} and s_{18}. The manager is holding ID $6 - 21$ in its list. Once s_6 is consumed, the manager will remove ID 6 and append ID 22.

Our streaming heuristic is based on the assumption that the network performance of each reader is predictable. The slice reader will request for new ID from the manager immediately after previous transfer is finished. The manager will assign a scheduled slice ID to the reader with the adaptive heuristic described in Algorithm 1. In this heuristic, the time to finish each slice is estimated with the prediction of reader's throughput. Thus the arriving order of the slices can be ensured by the preemption mechanism that a slow reader will be interrupted if a fast one becomes available. A skip mechanism is implemented to offset the low throughput of the slow reader and to improve the effective throughput of the entire system. Specifically, the *skipCount* of a reader will be increased by 1 when the reader is preempted. If the reader finishes a slice the counter will be decreased by 1 until it reaches 0. This counter is used to help slow reader skip proper number of immediately available slices, and thus to avoid unnecessary preemption.

Algorithm 1 Adaptive Slice Assignment Heuristic

INPUT:
L: the slice ID list.
R: the requesting reader
OUTPUT:
S: the ID of assigned slice.

```
1: count ← 0
2: for all unbuffered S ∈ L do
3:     count++
4:     if count > R.skipCount and S.estFinishTime > currentTime + S.remainingSize/R.throughput then
5:         if S.reader != NULL then
6:             S.reader.skipCount++
7:             S.reader.interrupt()
8:         end if
9:         S.reader ← R
10:        S.estFinishTime ← currentTime + S.remainingSize/R.throughput
11:        return S
12:    end if
13: end for
14: return NULL
```

7.3 Implementation Details

7.3.1 Asynchronous Execution

In our system, the native single-threaded task execution pattern is replaced by an asynchronous producer/consumer execution pattern to support multi-source streaming. The producer and consumer are coordinated with a locked buffer space (the circular buffer). Compared with the native execution pattern, this asynchronous implementation incurs two overheads: the cost of an extra memory buffer and locking overhead among threads. In practice, however, the impacts associated with these overheads are limited, since the size of buffer and the number of parallel threads are quite small.

We tested the time of reading one $128MB$ replica with the native method and the pipelined asynchronous method respectively. The time was measured inside the DFSInputStream object and reflected the gross cost of BlockReader reading a whole block. As shown in Figure 25, the single source reading performance can be improved by $8\% - 16\%$ on various platforms. It also shows that such improvement is related but does not contingent upon the availability of excessive CPU cores. We repeated the test on Cluster B with different configurations of CPU core per VM, and found

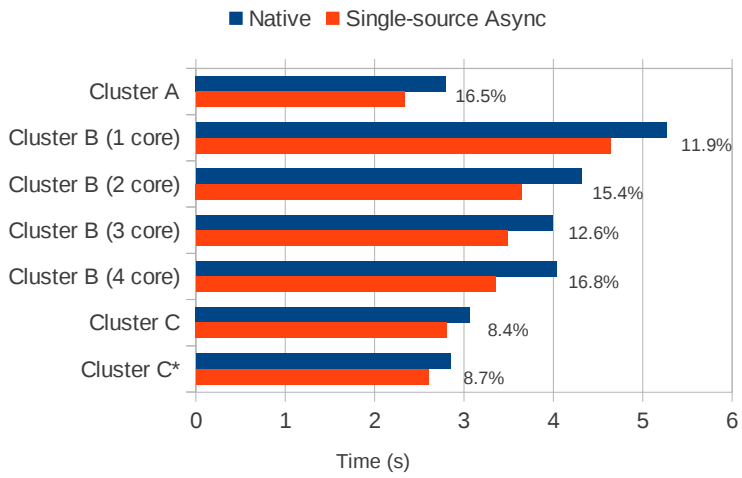

Figure 25: Performance of Asynchronous Execution

out that asynchronous execution can deliver noticeable improvement even with very limited CPU resource (1 core per VM). Similar behavior can also be observed in the comparison between Cluster C (single core EC2 instance) and Cluster C* (dual core EC2 instance).

7.3.2 Persistent Connection

The reading protocol of native Hadoop DataNode server is designed under the assumption of sequential data access, therefore only one section of continuous data can be requested and served within the lifetime of a network connection. In the multi-source streaming system, we introduced a new slice reading protocol to efficiently facilitate the pattern of accessing multiple discontinued subsections of data. Unlike the native protocol, in which server closes the connection after serving single request, the new protocol allows the server to maintain the connection and wait for new request from the same client. The new protocol is implemented with a unique operation code OP_READ_SLICE which guarantees the backward compatibility of the enhanced system.

The performance of three asynchronous reader implementations is compared with

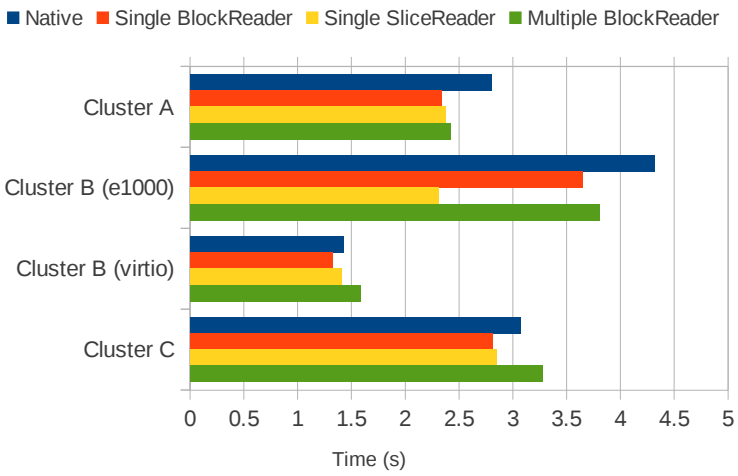

Figure 26: Performance of Different Reader Implementations

the native reader method in Figure 26. The time measured is also the gross time of reading a single replica. The red bar represents the time cost of using an asynchronous BlockReader to read a $128MB$ block, the yellow bar represents the cost of using an asynchronous SliceReader to read the block in 64 $2MB$ slices, and the green bar represents the cost of reading the slices by invoking BlockReader 64 times. The result shows that the persistent connection based SliceReader implementation performs uniformly better than the multiple-BlockReader implementation and only slightly worse than using a single BlockReader to read the whole block. Our evaluation also reveals an interesting behavior of kvm based virtual cluster (Cluster B). It seems that the widely-used $e1000$ NIC driver has some efficiency issue in supporting block-based data access of Hadoop system, as the slice-based data access turns out to be even faster than block-based access on Cluster B when the $e1000$ driver is configured. The performance relation becomes normal if the para-virtualized *virtio* NIC driver is used.

7.3.3 Throughput Prediction

The implementation of our adaptive streaming heuristic is a non-trivial task because of the difficulties in predicting the network performance. The throughput prediction is a critical building block of many network-related applications. Several advanced techniques, such as ARIMA [105] and neural network [40], have already been used in related studies to predict the network traffic. However, due to the busty nature of the network, none of these advanced techniques can substantially outperform simple moving average method in predicting the short time-scale LAN traffic as in the Hadoop cluster. Moreover, the sporadic traffic pattern in our system further complicates the prediction of a reader's throughput. Due to the limited buffer space, it is entirely possible that a reader has to wait for certain time between the transmissions of two slices. Such kind of voluntary traffic halt is neglected by most of the advanced prediction studies that presume continuous traffic.

Therefore, we decided to implement a basic predictor with the average throughput of effective transmissions in a fixed time window. Specifically, given the slice size x and window size w, if reader R has read a total of s slices and waited τ time inside the window, the prediction is made as $R.throughput = s \times x/(w - \tau)$. A reader will update its throughput prediction before every slice request. If no transmission takes place during the past w time, the old throughput value will be preserved. In our system, throughput prediction is also used by a background timer thread, which specializes in updating the estimated finish time of all the scheduled slices. Based on our observation, when a congestion takes place, a reader may be completely stalled in the lower level blocking read method for several seconds. In such circumstances, no one will be able to preempt the stalled reader, if the estimated finish time is not adjusted accordingly. By periodically adjusting the estimated finish time with a timer thread, the system can efficiently avoid the cascading stall problem when adapting to a congested network.

As an demonstration, we recorded the real-time throughput of our system on Cluster A with different congestion settings. In this test, a map task is reading a $128MB$ block from 3 remote replicas. The dynamic congestion is generated with the $iperf$ tool on the out-bound port of reader2's source node starting at 0.3 second. The result is plotted in Figure 27. Sub-figure (a) shows the congestion free setting, where the 3 readers contributes similar throughput. (b) shows the case of minor congestion, where reader2 is slowed down to about $10MB/s$ but the total throughput is unaffected due to the adaptation capability of our system. In case (c), an intensive congestion first stalls reader2, then the cascading buffer overrun further stalls the other two readers and greatly compromises the overall reading performance. In the last setting, a timer thread is configured to update the estimated finish time every 0.1 second. As shown in sub-figure (d), the stalled reader2 no longer causes buffer overrun, and the total throughput remains intact.

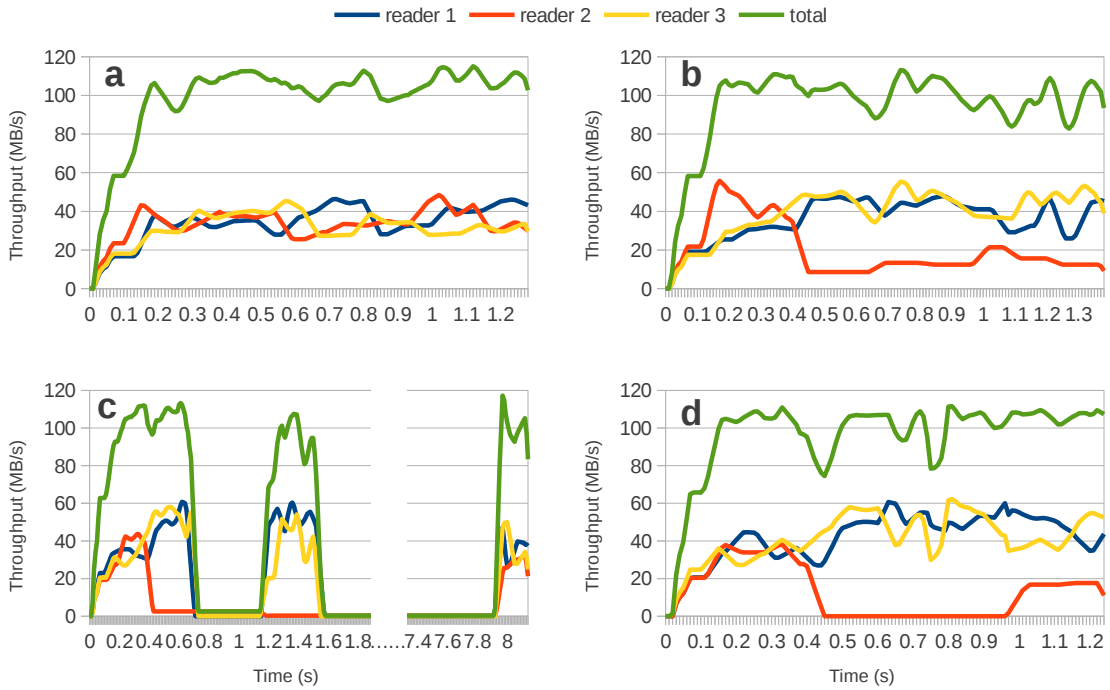

Figure 27: Real-time Throughput over Different Congestion Setups

7.3.4 Lazy Preemption

Since our throughput prediction algorithm is only capable of tracking slow variations in network performance, in practice a straightforward implementation of Algorithm 1 is going to suffer severely from false preemption. In the production system, a lazy preemption strategy should be implemented to accommodate a less predictable network. As shown in Algorithm 2, the assignment heuristic is modified in 3 ways: 1) the preemption condition is strengthened to "if the new reader can finish the slice 2 times faster than the current reader"; 2) the estimated finish time of a slice is calculated with full size rather than the remaining size of the slice; 3) preemption is considered only if there is no unassigned slice left in the pool.

Algorithm 2 Adaptive Slice Assignment Heuristic with Lazy Preemption

```
1: count ← 0
2: for all unbuffered S ∈ L do
3:     count++
4:     if count > R.skipCount and S.reader = NULL then
5:         S.reader ← R
6:         S.estFinishTime ← currentTime + S.size/R.throughput
7:         return S
8:     end if
9: end for
10: count ← 0
11: for all unbuffered S ∈ L do
12:     count++
13:     if count > R.skipCount and S.estFinishTime > currentTime + 2 × S.size/R.throughput then
14:         S.reader.skipCount++
15:         S.reader.interrupt()
16:         S.reader ← R
17:         S.estFinishTime ← currentTime + S.size/R.throughput
18:         return S
19:     end if
20: end for
21: return NULL
```

We evaluated the performance of three preemption strategies (No, Lazy, and Eager) on Cluster B by reading from 3 remote replicas. With the underlying OpenvSwitch's unique capability of accurate bandwidth control, we emulated different combinations of reader throughput including two balanced combinations and two imbalanced combinations. As clearly demonstrated in Figure 28, Lazy preemption has the most reliable performance, whereas the other two strategies are only good for part of the spectrum.

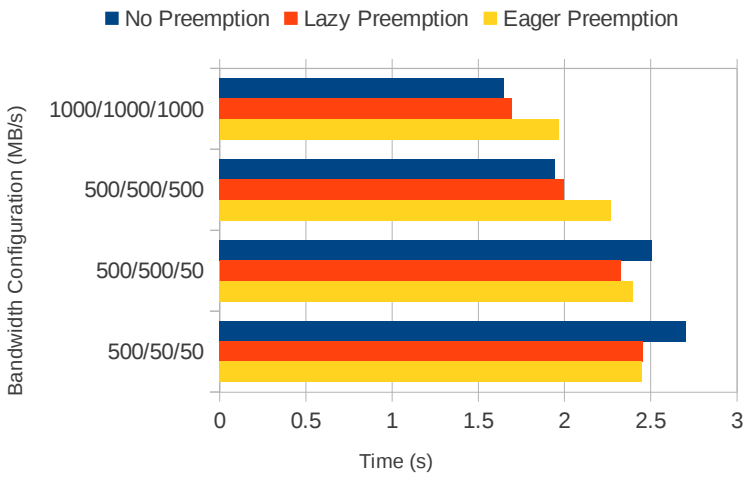

Figure 28: Performance of Different Preemption Strategies

7.4 Experimental Result

7.4.1 System Setup

In this research, 6 different clusters are used to evaluate the performance of our multi-source streaming enabled Hadoop system. In addition to the four clusters mentioned in the previous section, two heterogeneous clusters (E and F) are included to evaluate the new system against imbalanced environment. Both of them are made up of a combination of 16 EC2 medium instances and 16 EC2 small instances. The major difference between the two types of instances is that medium instance's network bandwidth is capped at around $1Gbps$ but small instance's network is capped at around $350Mbps$.

Table 6: Experimental Clusters

ID	N	Virt.	Network	Disk	Specification
A	4	No	Gigabit Ethernet	SATA HDD	Private Cluster
B	4	Yes	Configurable	SATA HDD	KVM Cluster
C	32	Yes	1Gbps	Instance Storage	EC2 mid. instance
D	32	Yes	1Gbps	EBS	EC2 mid. instance
E	32	Yes	1Gbps / 350Mbps	Instance Storage	16 mid. / 16 small
F	32	Yes	1Gbps / 350Mbps	EBS	16 mid. / 16 small

Our multi-source streaming enabled Hadoop system is implemented based on Hadoop 1.0.4. Unless stated otherwise, the experimental Hadoop cluster is configured as follows. JobTracker and NameNode service are setup on a dedicated master node. Each one of the slave node has one map slot and one reduce slot. The replication factor of HDFS is set to 2 due to the relatively small cluster size. Other major software environments and tools used in this study includes CentOS 6.4 (for the private cluster and VM host), Ubuntu 12.04 (for the virtual clusters), OpenJDK 1.6.0, OpenvSwitch 1.10.0.

The benchmarks used in the evaluation are Hadoop CopyToLocal, TestDFSIO, and TeraSort. The experiments are conducted with single-job multi-task execution pattern to precisely demonstrate the performance of our multi-source streaming enabled system. These experimental results are expected to be attainable on large-scale production clusters, where multiple jobs are competing for the resources, since the multi-source mechanism is oblivious to job scheduling policy and it merely introduce negligible excessive I/O on network and disk.

7.4.2 CopyToLocal Performance

CopyToLocal is a Hadoop shell command for copying file from HDFS to local file system. This command accesses the HDFS data blocks with exactly the same procedure as a normal Map task does, but it doesn't incur any task/job scheduling related overhead, so it can be used as a precise reading performance measurement. In this experiment, a $64MB$ file is copied from remote DataNode to the local file system.

The experiment results are shown in left part of Figure 29. In general, copy time is reduce by streaming simultaneously from two replicas. On cluster A and B, the throughput is almost doubled. On cluster C-F, the improvement is much smaller due to the limitation of their CPU performance. In Hadoop system, the reader has to verify the checksum of every remote data bit received. The EC2 instances used

in cluster C-F are configured with single-core CPUs, this prevents our streaming system from reaching its full capacity. To reinforce this observation, we also include Cluster C*(dual-core EC2 large instance) in this experiment. In the result of C*, the speedup of our method becomes much more significant. However, *extra CPU resources* are usually hard to find in most production clusters, and it is unfair if we are demonstrating speedups with systems of unlimited CPU power. Therefore, we decide to only use the low-performance cluster C-F in the following experiments to demonstrate the improvements that are achievable in realistic environments.

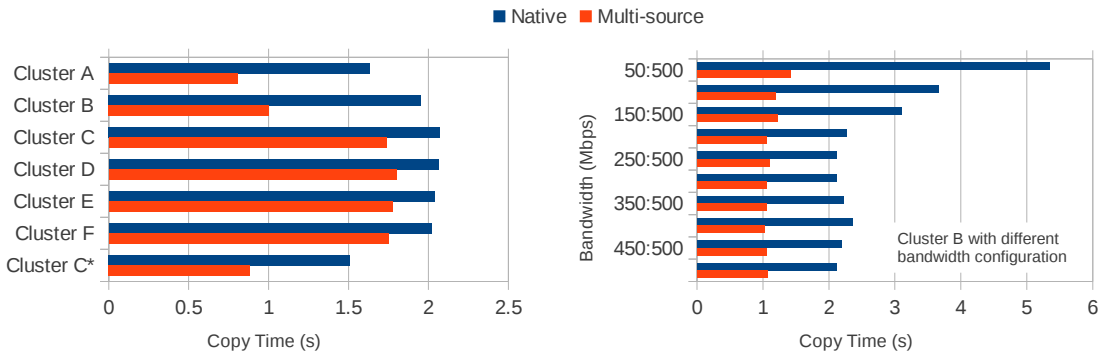

Figure 29: CopyToLocal Performace on Multiple Clusters

Another experiment conducted with the CopyToLocal command is the demonstration of multi-source streaming system's adaptation capability, as shown in the right part of Figure 29. The native system is unable to identify the bandwidth difference between the available sources, so its performance degrades drastically when a slow source is randomly selected. On the contrary, our multi source streaming system delivers much more consistent performance in the experiment. Under such imbalanced configuration, our system can achieve as much as 4× speedup.

7.4.3 TestDFSIO Performance

TestDFSIO is a standard benchmark included in the Hadoop distribution package. It can be used to evaluate the IO performance of the cluster or to stress the storage

system. TestDFSIO consists of two separate test options $-write$ and $-read$, in this experiment we are using the read test to emulate a dual-input application. The performance of our multi-source system is evaluated on the EC2 clusters (C-F). The data generated for the experiment are 32 $128MB$ files, each of which occupies exactly two HDFS blocks. Since TestDFSIO benchmark is designed to read each file with one map task, each map task in this experiment will access two blocks resembling the pattern of applications such as PageRank and database join operation.

The most important metrics reported in TestDFSIO are the Throughput and the Average IO Rate. Both of them are indication of HDFS's reading performance. As shown in Figure 30, the multi -source streaming system improves both metrics by 10% to 20%. Among the different cluster configurations, the new system can achieve higher speedup on slower EBS storage and imbalanced network due to the greater performance degradation of native system in these cases.

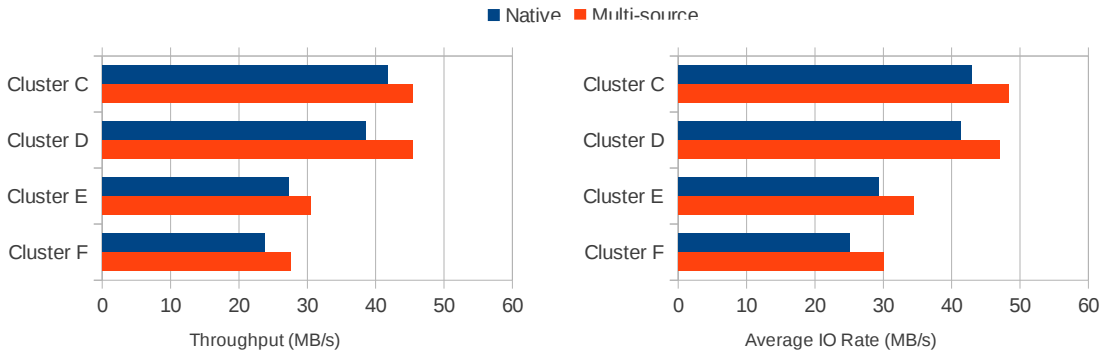

Figure 30: TestDFSIO Performance on Multiple Clusters

7.4.4 TeraSort Performance

TeraSort is a well-known Hadoop benchmark developed by Yahoo!. This benchmark is able to examine the overall performance of a Hadoop system by stressing both the MapReduce and the HDFS subsystems. The entire benchmark flow consists of three jobs: TeraGen for generating random input data, TeraSort for sorting the data,

and TeraValidate for validating the sorted data. Our evaluation is focused on the TeraValidate part, since the other two jobs have little need for remote data. In this experiment, 8 map tasks are launched to validate $10GB$ sorted data. The average execution time of map task are listed in Figure 31. According to the result, our multi-source streaming system has achieved about 15% speedup on all the four platforms.

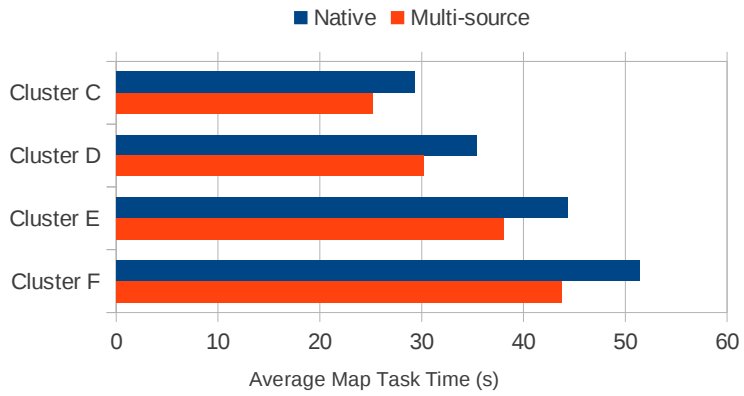

Figure 31: TeraValidate Performance on Multiple Clusters

7.4.5 Summary

According to the experimental results, the multi-source streaming mechanism is expected to improve the throughput of a Hadoop cluster in following scenarios.

- *In-rack job execution.* In-rack job execution is the major case where read traffic takes place. Previous study shows that in-rack assignment can be satisfied for almost 100% of the map tasks. Due to the fact that HDFS will always create two replicas in the same rack and create the other replica elsewhere, there is a 50% chance for the system to stream input data simultaneously from two fast in-rack sources and thus achieve larger bandwidth.

- *Dual-input application.* Since Hadoop system can only explore locality for one of the two inputs, it is likely that the map performance of dual-input applications will be limited by the remote input. By streaming the remote data from multiple

sources, the multi-source streaming mechanism can effectively speed up the whole task.

- *Congested resources.* In Hadoop system, the tasks are assigned to slots, and the number of slots are proportional to available CPU cores. The network and storage resources of each node are usually shared by multiple map slots and reduce slots. In some cases, one source node can suffer serious congestion and yield a throughput that is significantly smaller than its peers. The multi-source method will adaptively avoid this congestion by streaming more data from the alternative sources.

- *Virtualized cluster.* The multi-source streaming mechanism is expected to be most suitable for the virtualized Hadoop clusters, in which basic system characteristics (such as CPU performance, disk locality, network bandwidth, and system topology) are no longer static. In these clusters, the ability to adapt to the unspecified or even time-variant network infrastructures can effectively boost the throughput of remote data access.

7.5 Related Works

The availability of multiple data sources has been partially exploited by native Hadoop system during the shuffle phase. The reduce tasks are capable of downloading multiple shuffle files simultaneously. In this multi-source transfer scenario, files on different source nodes are downloaded by a small pool of threads without particular downloading order. The reduce task starts processing once all the files are ready. Such design is infeasible for reading input files, because the map task assumes a streaming read pattern. Although the MapReduce model guarantees that the input of splittable key-value pairs can be processed in arbitrary order, the underlying storage system should be accessed with streaming I/O interface as well, because the storage system itself is agnostic to the boundary of logical entries.

A few studies have been done to exploit the availability of multiple input sources and comply with the streaming requirement. In the pre-fetch and pre-shuffle research [112], the intra-block prefetch method can read from two replicas in reverse orders. However, their method can not utilize more than two replicas. Moreover, they are using a buffer as large as the input block, which significantly increases memory requirement of each task. To the best of our knowledge, our work is the first attempt to enable adaptive multi-source data streaming in Hadoop.

The topic of multi-source streaming has been investigated in a number of studies under the context of media content distribution, especially in peer-to-peer systems. For example, rate allocation and packet partition algorithms are developed in [92] to minimize the packet loss rate and the probability of late packet arrivals. A media data assignment algorithm is proposed in [139] to reduce buffering delay. A dynamic rate allocation and packet partition scheme is proposed in [77] to adapt to the sender's varying throughput. An adaptive layered streaming algorithm is proposed in [4] to compensate for variations in the measured bandwidth from senders.

The techniques used in these systems are not directly applicable to Hadoop clusters, since the design objectives for the two kinds of systems are different. Most media oriented systems are designed for continuous content playback, so their objectives are to minimize initial buffering delay or to control packet loss rate. In Hadoop system, where initial delay is not a concern and packet loss is intolerable, the objectives usually focus on throughput or utilization.

CHAPTER VIII

MULTICAST-BASED REPLICATION ON HADOOP CLUSTER

In this chapter we present the technique of improving the remote write throughput of Hadoop by enabling multicast-based replication. This new mechanism is based on a congestion-controlled reliable multicast socket (CCRMSocket), which leverages the multicast feature of network switches to reduce the network traffic of data replication. We first discuss the challenges and opportunities of developing such multicast functionality for Hadoop cluster, and then present the design of CCRMSocket in detail. The performance of CCRMSocket is evaluated with our multi-rack experimental cluster. The impact of multicast-based replication is further studied using a Hadoop cluster simulator and a workload trace obtained from large-scale production system.

8.1 Challenges and Opportunities

In computer network, multicast is the mechanism for one-to-many data delivery. IP multicast is the most widely supported multicast implementation on Ethernet. It sends datagrams to a group of destinations in a single transmission. A special range of IP addresses (224.0.0.0 - 239.255.255.255) are used to identify the multicast groups. A receiver can dynamically join or leave any multicast group by sending IGMP (Internet Group Management Protocol) packet to the designated address. The IP multicast itself only supports best-effort transmission of unreliable UDP datagrams. Congestion-controlled and reliable protocols can be implemented on top of IP multicast.

Due to the one-to-many communication patten, multicast is born with the intrinsic conflict between the scalability (which can suffer from frequent feedback) and the congestion-controllability (which requires real-time feedback). Because of its far-reaching scope, multicast also creates challenges for network security and stability. A problematic or evil-minded multicast traffic flow can seriously disrupt the entire network. Consequently, it is a daunting task to create a full-featured multicast implementation for a general network environment such as the Internet. Although great research efforts have been devoted to the development of congestion-controlled and especially TCP-friendly multicast protocol, successful high-performance multicast applications are limited.

The design of typical Hadoop cluster brings us a unique set of opportunities making the implementation of our high-performance CCRMSocket possible.

1) The default replication factor of HDFS is three. Such a small set of destinations significantly relaxes the scalability requirement of multicast, and thus breaks the conflict between scalability and congestion-controllability. With the real-time feedback coming from all the destinations, the CCRMSocket can precisely adjust the transmitting rate to coexist with other network traffics.

2) The write operations in Hadoop cluster are administrated by the centralized HDFS NameNode and then fulfilled by the distributed DFSClient and DataNodes. This design makes it possible to securely implement the multicast functionality inside HDFS. Hiding multicast functionality from the user not only guarantees backward compatibility of the system, but also prevents the user from accidentally flooding the network with problematic multicast flows. Moreover, such isolation also provides opportunity for future security hardening.

3) Typically, a Hadoop cluster is connected with a balanced hierarchical network, and the computer nodes are configured with homogeneous hardware and software. Such a symmetrical topology is great for the worst-case performance of multicast

communication since the differences in bandwidth latency and processing power are potentially minimized across any group of multicast receivers.

8.2 System Framework

To enable multicast-based replication, two new components are developed in Hadoop HDFS: the multicast group manager and the reliable congestion-controlled multicast socket. The NameNode, the DataNode and the DFSClient of native HDFS are also modified to interact with the new components.

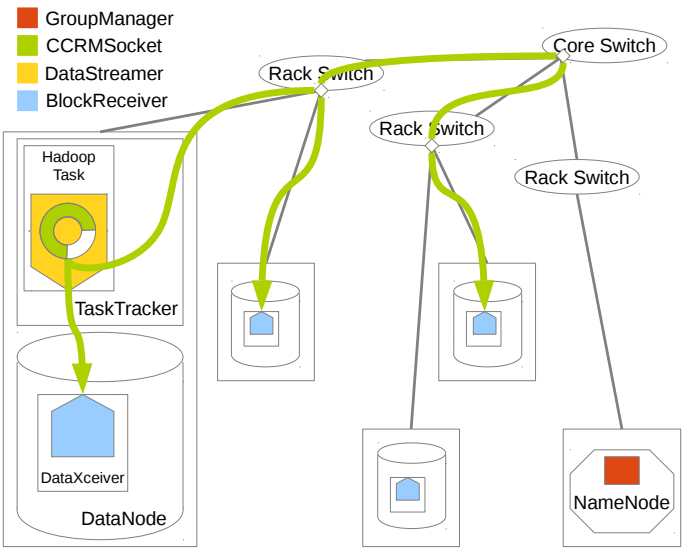

Figure 32: Multicast-enabled Hadoop System

Under the hood of native Hadoop, the data written by MapReduce tasks are delivered by the DFSClient's DataStreamer and then relayed by the DataNode's BlockReceivers. To make our system backward-compatible to existing Hadoop systems, the pipelined TCP connections linking DataStreamer and the BlockReceivers are preserved alongside with the new CCRMSocket connection. When conducting multicast replication, the DataStreamer simply connects to DataNode with a new opcode "OP_CAST_BLOCK" instead of the original "OP_WRITE_BLOCK" to notify the BlockReceiver that data will be delivered via multicast socket and relaying is unnecessary for this block.

The multicast group manager (GroupManager), which is designed as a part of the NameNode, controls the pool of multicast group address. In reply to the DFSClient's block write request, the NameNode assigns a pair of multicast group address and port number in addition to the set of the DataNodes that will store the replicas. The address is recycled back to the GroupManager's pool when the write operation terminates. Theoretically there are 2^{28} unique multicast group addresses, but the actual size of the pool should not exceed the maximum number of concurrent multicast connections that are supported by the network switches. When the pool is exhausted, the NameNode can reply an empty multicast group address and let the DFSClient fall back to pipelined replication gracefully.

The congestion-controlled reliable multicast socket (CCRMSocket), which combines the unreliable IP-multicast with a TCP-style window-based congestion control algorithm, forms the backbone of the our system. To maintain the portability of Hadoop, the CCRMSocket is developed entirely with native Java MulticastSocket (used as data channel) and Java DatagramSocket (used as acknowledgement channel). It also provides the standard OutputStream interface to the Hadoop tasks on the source-side and the InputStream interface to the DataNodes on the destination-side. Circular buffers are introduced to both sides of CCRMSocket to enable asynchronous network IO. Similar to the design of TCP, the congestion-control function of CCRMSocket is implemented on the source buffer to adjust the rate of data transmission.

The new replication method requires the network switches in the cluster to support IGMP, and this may potentially limit the compatibility of multicast-enabled Hadoop. However, we argue that multicast-enabled Ethernet switches are widely available today. IGMP support has been offered for a long time by major network vendors in both high-end and low-end product lines of their manageable switches. Consequently, such capability could already be embedded in many existing Hadoop clusters. Multicast gets worse support in cloud environment. Currently, only a fraction of service

providers (e.g. Dimension Data) support IP multicast and IGMP. However, with the development software-defined network technology, major providers such as RackSpace are actively preparing their multicast integration.

8.3 CCRMSocket Design

As shown in Figure 33, the Hadoop task feeds data into to the CCRMSocket as an OutputStream writer, and the DataNode pulls data out as an InputStream reader. To improve the throughput, additional threads are spawned by the socket to exchange data and acknowledgement (ACK) asynchronously. The auxiliary threads include a multicast sender and an ACK listener on the source, as well as a multicast receiver and an ACK responder on each of the destinations. The main writer/reader thread interact with the auxiliary threads through a circular buffer that holds intermediate data.

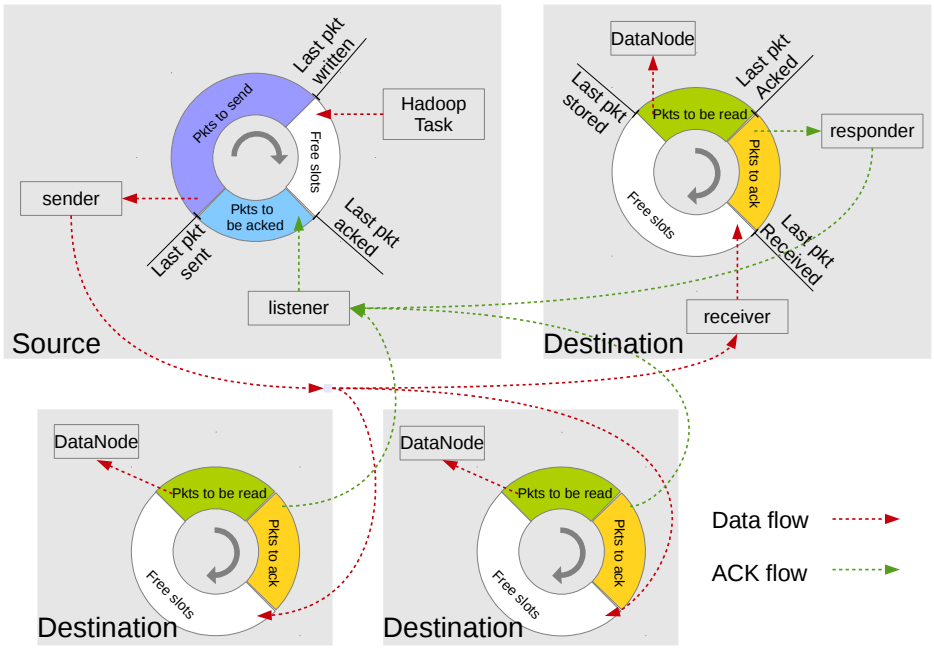

Figure 33: Data flow inside CCRMSocket

On the source side, the writer assembles the data into fixed-size packets and appended them into the buffer. Each of these packets carries a unique sequence

number, several flag bits, and a chunk of data as payload. The writer can be blocked by a saturated buffer or by the congestion control algorithm. The sender runs behind the writer sending out the packets one-by-one through the MulticastSocket. The listener processes the incoming ACK packets, and frees up the buffer slot in which the residing packet has been fully acknowledged. On the destination side, the multicast receiver, the ACK responder and the InputStream reader are operating with a similar pattern. The receiver injects the multicasted data packets into the circular buffer. After receiving each packet, the responder replies to the source's DatagramSocket with an ACK packet carrying the sequence number of the most expected packet (the first packet that has not been received) and the responder ID. The reader sequentially consumes the data and frees up the buffer slot. If the next packet has not been received then the reader is blocked. Alternatively, if the reader does not consume fast enough, the receiver can be blocked by a saturate buffer.

A fundamental design choice to make in the development of the CCRMSocket is the transmission granularity. The primitive transmission unit we can manipulate in Java is the DatagramPacket, which is essentially a wrapper on the basic IP datagram. Although the size of an IP datagram can be as large as 64KB, it is actually constrained by the data-link layer beneath it. The underlying network may quietly split the datagram into fragments to meet its own transmission granularity. If any one of these fragments gets dropped in route, the entire datagram is corrupted. Consequently, a smaller datagram is less vulnerable to the fragmentation problem, but it does increase the transmission overhead. Since the user-level implementation can make our CCRMSocket more prone to the overhead problem, we set the size of DatagramPacket to 1458-byte (4 bytes sequence number + 4 bytes reserved + 1450 bytes data) which is close to the Ethernet MTU of 1500 bytes. The circular buffer we used in CCRMSocket has 512 slots, so the total transmission buffer space is 725KB.

8.3.1 Congestion Control Algorithm

The congestion-control algorithm inside the CCRMSocket is designed based on TCP-Reno. The transmission state is determined by two major variables on the source: the congestion window ($cwnd$), which limits the number of packets in flight; and the slow-start threshold ($ssthresh$), which throttles the opening rate of congestion window. The window grows exponentially in slow-start phase and linearly in congestion-avoidance phase. A packet is deemed as lost if multiple duplicated acknowledgements are received or if no acknowledgement has been received in a prolonged period. Suppose lpw represents the sequence number of the latest packet that has been written, lps represents the latest packet sent, and lpa represents the latest packet acknowledged. Given that the source buffer has s slots and the data is sent to r destinations, the algorithm can be expressed as follows.

At the beginning of the multicast session,

$$lpw = lps = lpa = 0; \tag{13}$$

$$ssthresh = s; \tag{14}$$

$$cwnd = r. \tag{15}$$

The writer can write the next packet, if

$$lpw - lpa < s. \tag{16}$$

The sender can send the next packet, if

$$lpw - lpa > 0 \text{ and } lps - lpa < cwnd. \tag{17}$$

On the reception of a new acknowledgement, the congestion window is opened as

$$cwnd = \begin{cases} cwnd + 1/r & \text{if } cwnd < ssthresh \\ cwnd + 1/r/cwnd & \text{otherwise.} \end{cases} \tag{18}$$

On the reception of $3r$ duplicated acknowledgements, the expected packet is retransmitted immediately and the state variables are adjusted as

$$ssthresh = cwnd = cwnd/2. \tag{19}$$

If the listener receives nothing in 10 milliseconds, it starts retransmission by resetting the state variables as

$$lps = lpa; \tag{20}$$

$$ssthresh = s; \tag{21}$$

$$cwnd = r. \tag{22}$$

In CCRMSocket, the window opening rate and the fast retransmission trigger are implemented differently from TCP-Reno's due to the fact that one multicasted packet can appear lost to more than one destinations. We delayed fast retransmission slightly to mitigate the impact of such multiplied losses.

The retransmission time out (RTO) in CCRMSocket is also implemented differently. In TCP-Reno, the RTO is adjusted according to the average round trip time (RTT) of the packets. That design was abandoned, because it is inefficient for us to either precisely measure the RTT or constantly adjust a timer at the user-level. Alternatively, we set up a 10 millisecond timeout in the ACK listener's DatagramSocket, such that the transmission is restarted if the listener receives nothing in 10 millisecond. Since the CCRMSocket is designed for the Hadoop clusters, in which the network condition is much more predictable than those general environments faced by TCP, this fixed RTO implementation should be acceptable.

8.3.2 Synchronization

Since the multicast socket is implemented with multiple threads, their accesses to the shared circular buffer should be coordinated. There are three ways to implement such coordination: lock-free, fine-grained locking, and coarse-grained locking. The

lock-free method, in which the threads are spinning on per-slot condition variables, can theoretically achieve the best performance but wastes too much CPU cycles. The locking method, either per-slot-lock-based or centralized-lock-based, uses Java's wait-and-notify mechanism and are more efficient. We have tested all these alternatives and decided to implement the CCRMSocket with a centralized lock over the entire buffer, because we find such method is fast enough to saturate a Gigabit Ethernet network connection.

8.3.3 CPU utilization

When operating at the line rate, the CCRMSocket does impose noticeably higher load on CPU than the TCP normally does because all the data and acknowledgements (about 160,000 packets per second) are handled in user-level. With multiple CPU-load-alleviating techniques, such large packet size, lock-based synchronization, and simplified retransmission timer, we managed to lower the CPU load to less than 150% (or 1.5 cores) on our test platform. We expected the CPU load to be reduced considerably if some of the multicast functionalities can be implemented inside the operation system.

8.4 Experimental Result

The performance of CCRMSocket is evaluated on our multi-rack test platform. The platform consists of four racks with six nodes in each rack. All the nodes are equipped with a quad-core Intel i5-2500 CPU, 8 GB memory, and 250GB hard drive. A TP-Link TL-SG3210 GbE switch is used in each rack. The four rack switches are then connected to a Cisco SG300 GbE switch. The software stack includes CentOS 6.5, Openjdk 7, and Hadoop 1.2.1. The experiment also uses Iperf 2.0.5 to generate TCP traffic.

As demonstrated in Figure 34, the multicast sender is located on node N1, the three receivers are located on N2 N3 and N13 respectively. This experimental setting

resembles HDFS's data placement heuristic that puts two replicas at the local rack and one replica at a remote rack.

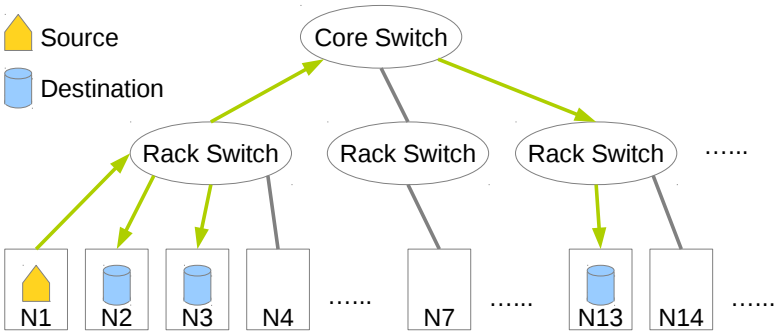

Figure 34: Experimental Multicast Setting in Multi-rack Cluster

When there is no other traffic in the network, we measured a stable multicast throughput of $104.7MB/s$, which matches the optimal TCP throughput on Gigabit Ethernet. Then, we created a coexist TCP flow to evaluate the TCP-friendliness of our multicast implementation. We tested the CCRMSocket under various congestion patterns by selecting different source and destination for the TCP flow. Table 7 lists the measured multicast throughput (MB/s) and its deviation from the expected throughput (positive means faster, negative means slower). The CCRMSocket exhibits less than $\pm 25\%$ deviation in all test cases. We also measured the packet retransmission rate, which is less than 1% in all cases.

The experimental result shows that 1) the CCRMSocket can save node-to-switch bandwidth; 2) the CCRMSocket is faster enough to saturate a Gigabit Ethernet connection; 3) although the CCRMSocket is not strictly fair to TCP, it can peacefully coexist with TCP traffic; 4) the CCRMSocket does not litter the network with wasteful retransmissions.

8.5 Simulation Studies

A production Hadoop cluster can span across hundreds or even thousands of nodes and process enormous amount of jobs simultaneously. Although our test platform

Table 7: Multicast Throughput With Coexisted TCP Flow

TCP flow		Multicast throughput		TCP flow		Multicast throughput	
src	dst	measured	deviation	src	dst	measured	deviation
N1	N2	61.8	19%	N4	N14	58.3	12%
N1	N3	64.3	24%	N7	N1	101.8	-2%
N1	N4	64.4	24%	N7	N2	57.9	11%
N1	N7	62.0	19%	N7	N4	100.3	-4%
N1	N13	64.7	24%	N7	N13	48.4	-7%
N1	N14	65.7	26%	N7	N14	48.0	-8%
N2	N1	99.5	-4%	N13	N1	94.1	-10%
N2	N3	46.1	-11%	N13	N2	54.5	5%
N2	N4	91.4	-12%	N13	N4	87.8	-16%
N2	N7	48.1	-8%	N13	N7	89.7	-14%
N2	N13	51.5	-1%	N13	N14	87.5	-16%
N2	N14	54.5	5%	N14	N1	101.9	-2%
N4	N1	101.1	-3%	N14	N2	54.2	4%
N4	N2	58.3	12%	N14	N4	101.2	-3%
N4	N7	55.7	7%	N14	N7	88.5	-15%
N4	N13	58.7	13%	N14	N13	57.6	11%

is capable of evaluating CCRMSocket's performance, we are unable to conduct a large-scale experiment with it to demonstrate the potential impact of multicast on production systems due to the limited cluster-size and the lack of workload. Alternatively, we developed the HFlowSim simulator to study the big question of how does multicast-replication benefit a Hadoop system. Based on our workload model that simplifies the Hadoop jobs into data flows, HFlowSim simulates the system by scheduling the flows and updating their throughputs. The job trace used in HFlowSim is gathered from Facebook's 3000-node production cluster and published in the SWIM project [26]. This trace contains more than 9000 jobs and has a span of 24 hours.

We did not use existing Hadoop simulators [58, 85, 74] because the majority of them are developed to study job scheduling or resource management mechanisms, and are unable to simulate multicast network traffic. The ns-2 based MRPerf [127] is the only candidate that can simulate the network traffic of Hadoop system in detail. However, the packet-level simulation granularity of ns-2 makes MRPerf prohibitively

slow in simulating large-scale Hadoop system.

8.5.1 Simulation Design

In HFlowSim the Hadoop cluster is modelled as a 3-level hierarchical network graph with directed edges. The bottom-level nodes in the graph represent computers, the middle-level nodes represent rack switches, and the single top-level node represents the core switch. Each edge in the graph represents a directed network path and has a fixed bandwidth capacity. The network traffic is modelled as flows across the computer nodes. We use three types of flow to simulate all kinds of traffic going on in a real Hadoop system. The single-destination flow represents the TCP traffic in data shuffling between map/reduce tasks, the pipelined flow represents the TCP traffic in native replication mechanism, and the multicast flow represents our new replication mechanism. Each flow comes with a predefined amount of data as its load, and the load is delivered gradually with a time-variant throughput. At any moment, a flow consumes same amount of bandwidth along all the edges through which it travels. A flow terminates once all the load is delivered.

The simulation is conducted in discrete time ticks, each of which represents 1 millisecond of wall time. All network flows are updated at every time tick following the additive-increase/multiplicative-decrease (AIMD) rule of TCP traffic. Specifically, the throughput of a flow will be either increased by 1MB/s if all the network paths it travelling through has 1MB/s residual capacity available or decreased by half if any of the paths is saturated. Such increment or decrement takes effect immediately along the edges so that the peer flows (those who share at lease one edge with this flow) can be updated accordingly in the rest of this time tick. At the beginning of each time tick, the sequence of all the flows are shuffled with the Knuth algorithm to make sure that each flow has the same probability to be updated first among its peers. Without doing per-packet calculation HFlowSim tracks the arrivals and terminations

of flows to update the bandwidth distribution among the dynamic network. The same simulation technique has also been used in existing research [7] to study data center network.

We model the workload as a series of Hadoop jobs and each job as a set of reduce tasks. Every task in our simulation has two consecutive execution stages: the read stage and the write stage. In the read stage, which resembles that data shuffling of a Hadoop job, the task reads data via multiple single-destination flows from different computer nodes. In the write stage, which resembles the creation of data replicas on multiple HDFS nodes, the task writes data via one multi-destination flow to different computer nodes. When a job is arrived, all of its tasks are appended into a cluster-wide queue. The task at the head of the queue will be scheduled to execute once an idle computer node is available. One computer node can only host one task at a time, but can be read from and/or written to by remote tasks simultaneously. Such mechanism complies with the FIFO scheduling policy used by real Hadoop system in scheduling the reduce tasks. The complete simulation procedure is listed in Algorithm 3.

In real Hadoop system, a job typically consists of map tasks and reduce tasks. We deliberately excluded all the map tasks from the simulation to rule out all the performance factors that are less relevant to the network, such as input data locality and map task scheduling. According to previous research papers [141], most of the map tasks in real world read from and write to their local hard disks. They produce very little network traffic and are consequently less dependent on the performance of the network. The reduce tasks, on the contrary, are very sensitive to the network, as they always read from and write to remote computer nodes. We further excluded the computation cost and the disk IO cost of the tasks from the simulation, not only because such detail is not specified in the workload trace we are using, but also because they are orthogonal to the communication time cost that we are focusing on currently.

Algorithm 3 Simulation Procedure of HFlowSim

```
1: import job trace into eventQueue
2: tick ← 0
3: while eventQueue ≠ ∅ OR flowList ≠ ∅ OR taskQueue ≠ ∅ do
4:     if flowList ≠ ∅ then
5:         shuffle flowList
6:         for all f ∈ flowList do
7:             if f.load > 0 then
8:                 adjust f.throughput
9:                 adjust network bandwidth
10:                f.load ← f.load − f.throughput
11:            else
12:                adjust network bandwidth
13:                remove f from f.task.flows and flowList
14:                if f.task.flows = ∅ then
15:                    create a TaskEvent in eventQueue
16:                end if
17:            end if
18:        end for
19:    end if
20:    if eventQueue ≠ ∅ then
21:        for all e ∈ eventQueue AND e.time = tick do
22:            if e = JobEvent then
23:                insert tasks into taskQueue
24:            else if e = TaskEvent then
25:                if e.task.stage = read then
26:                    e.task.stage ← write
27:                    add write flow into e.task.flows
28:                    add e.task.flows into flowList
29:                else if e.task.stage = write then
30:                    add e.task.node into nodeList
31:                end if
32:            end if
33:        end for
34:    end if
35:    while taskQueue ≠ ∅ AND nodeList ≠ ∅ do
36:        remove t from taskQueue
37:        remove n from nodeList
38:        t.stage = read
39:        t.node = n
40:        add read flows into t.flows
41:        add t.flows into flowList
42:    end while
43:    tick + +
44: end while
```

8.5.2 Simulation Result

In the simulation, we tested three different cluster configurations: C16x32x1 is a 16-rack cluster with 32 node-per-rack and 1 task-per-node; C8x32x1 is a 8-rack cluster with 32 node-per-rack and 1 task-per-node; C8x32x1 is a 8-rack cluster with 32 node-per-rack and 2 task-per-node. We fixed the node-to-switch bandwidth of all the cluster to $1Gbps$, and then altered the switch-to-switch bandwidth from $10Gbps$ to $40Gbps$ to simulate different over-subscription ratio in the backbone network. Due to the difference in scale, the three clusters exhibited different utilization rate under

the same input workload. In average, C16x32x1 is about 40% occupied, C8x32x1 is about 75% occupied and C8x32x2 is about 60% occupied.

The performance metrics we studied are the execution time of each job, each task, and each write stage. The write time can directly show the impact of multicast. The task time is the summation of the time used in read stage and write stage of each task. The job time is measured from the arrival of the job to the end of its last task, so it depends not only on the execution time of the tasks but also on the queuing time of the tasks. We ran the simulations with multicast-based replication and pipelined-based replication respectively, and measured the differences on the metrics.

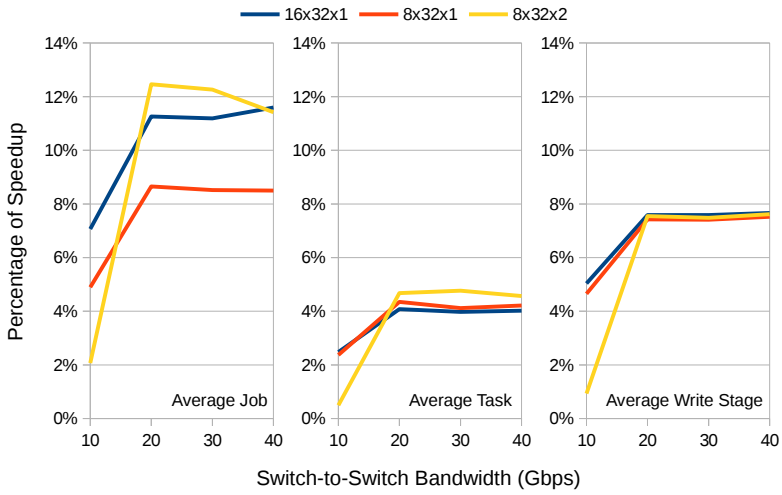

Figure 35: Multicast Speedup on Average Job, Task, and Write Stage

Figure 35 shows the speedup of multicast on different metrics. In the best case, multicast-based replication can accelerate the writes by 8% and accelerate the jobs by 12%. The result indicates that 1) multicast can systematically accelerate a Hadoop cluster, and 2) the benefit is larger if the cluster is less busy. Network over-subscription on either the node-to-switch paths or the switch-to-switch paths may undermine the potential of multicast.

In Figure 36, we counted the number of jobs on cluster C16x32x1 that are accelerated or decelerated in terms of average task time, and then plotted the accumulated

distribution against the total task time of the job. The result reveals that multicast benefits the tasks of long jobs more than the tasks of short jobs. In Figure 37, we plotted the percentage of speedup on average write time of each job against the total output size of the job. As shown in the figure, the jobs with heavier load of data to write usually have higher chance being accelerated by multicast-based system. According to these detailed simulation result, it tends out only the big jobs are directly accelerated by multicast. The majority of the smaller jobs are accelerated because their queuing time are shortened when the big jobs run faster. This also explains why the average speedup on jobs is higher than the average speedup on tasks.

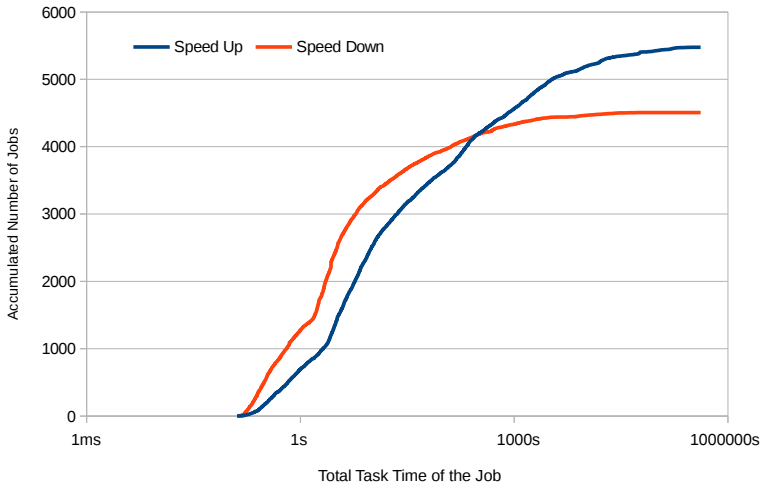

Figure 36: Accumulated Number of Accelerated and Decelerated Jobs

8.5.3 Summary

According to the experimental result and the simulation study, we are confident to conclude that the multicast-base replication is a feasible alternative to the native pipeline-based approach in native Hadoop system. Our current user-level implementation of CCRMSocket has demonstrated the great potential in saving network bandwidth. The simulation over real workload trace further confirms that the multicast-base replication can effectively increase the throughput of a large-scale production

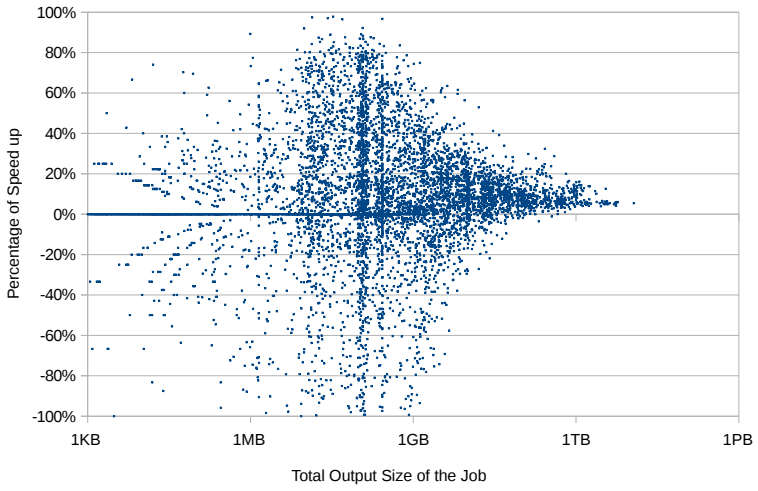

Figure 37: Experimental Multicast Setting in Multi-rack Cluster

Hadoop system. Since the performance of our CCRMSocket is directly limited by the constant switching between privileged space and unprivileged space, significant improvement can be expected if the congestion-control functionalities are support in the operating system.

8.6 Related Works

Existing reliable multicast protocols include SRM [45], RMTP [83], PGM [46], and NORM [3]. In these protocols feedback loop is created to detect lost and/or out-of-order messages and to take corrective actions. For the reliable multicast transmission to scale up, the protocol should prevent the multiplied feedback messages from overloading the sender. To achieve this goal, RMTP uses a tree-based acknowledgement (ACK) mechanism, in which the positive feedback messages from receivers are first aggregated by hierarchical representatives and then delivered to the multicast sender. The other three protocols are based on a negative acknowledgement (NACK) design, in which the feedback is sent only when the receiver detects one or more lost packets. To further reduce the number of feedback messages, both SRM and NORM insert

random back-off before sending out NACK with the expectation that the absent packets are NACKed by other peer receiver already. PGM uses a tree-based mechanism similar to RMTP, but it relies on specially designed switches and routes to aggregate the NACKs.

Congestion-control is a critical capability for any multicast protocol to efficiently operate on a busy network. Most of the experimental implementations use the rate-based approach, in which the transmission rate is explicitly adjusted according to the network feedbacks that indicate congestion. For example, MDPCC [88] uses a steady state TCP throughput model to estimate the rate of a TCP flow under an equivalent network condition. TFMCC [132] let the receivers determine the desirable receiving rate that is TCP-friendly based on their own observations of round trip delay and packet loss rate. Alternatively, PGMCC [108] uses a TCP-style window-based congestion control approach. The main idea of PGMCC is to select the slowest receiver from the group and let this receiver acknowledge each packet (while others only send NACK), such that the sender can adjust its congestion window according to the slowest receiver. Due to the small replication factor of Hadoop HDFS, such selection of the slowest receiver is not needed in the implementation of CCRMSocket. However, limiting the source of ACK packets to the slowest receiver can potentially reduce the CPU load of feedback listener.

To the best of our knowledge, this research is the first systematic attempt to evaluate the integration of congestion-controlled multicast into Hadoop. Existing projects related to this research are JetFile [51] and JGroups [17]. JetFile is a multicast-based distributed file system, in which the SRM protocol [45] is used to locate metadata and to create data replicas. The practicality of JetFile is impaired by the limited performance of SRM. JGroups, an open-source Java library for reliable group messaging, supports both TCP-based and multicast-based delivery methods. JGroups can limit

the transmitting rate statically when multicast is used. TCP-friendly congestion-control is supported in neither of these projects.

CHAPTER IX

SUMMARY

Traditional cluster computing systems such as the supercomputers are equipped with specially designed high-performance hardware, which escalates the manufacturing cost and the energy cost of those systems. Due to such drawbacks and the diversified demand in computation, two new types of clusters are developed: the GPU clusters and the Hadoop clusters. The GPU cluster combines traditional CPU-only computing cluster with general purpose GPUs to accelerate the applications. Thanks to the massively-parallel architecture of the GPU, this type of system can deliver much higher performance-per-watt than the traditional computing clusters. The Hadoop cluster is another popular type of cluster computing system. It uses inexpensive off-the-shelf component and standard Ethernet to minimize manufacturing cost. The Hadoop systems are widely used throughout the industry.

Alongside with the lowered cost, these new systems also bring their unique challenges. According to our study, the GPU clusters are prone to severe under-utilization due to the heterogeneous nature of its computation resources, and the Hadoop clusters are vulnerable to network congestion due to its limited network resources. In this research, we are trying to improve the throughput of these novel cluster computing systems by increasing the workload parallelism and communication parallelism.

In the GPU clusters, although both CPU and GPU are capable of doing computation, these two independent processing units are suitable for different workloads. To achieve a better performance, the application developers should carefully decide which part of their job is executed on CPU and which part is on GPU based on to the characteristics of the workload. On the other hand, however, the hardware

CPU-to-GPU ratio of any cluster is fixed at the time of deployment. The intrinsic mismatch between the workload-dependent demand and the static availability of such heterogeneous resources is preventing the GPU clusters from being efficiently utilized. In this research, we are focusing on increasing the workload parallelism by enabling cluster-wide GPU sharing and heterogeneous job collocation. A dynamic GPU workload dispatch framework is developed to improve the sharing of GPU devices among different jobs. A job collocation system is also developed to even out the mismatch and boost the utilization of CPU resources.

In the Hadoop clusters the limited bandwidth is a major bottleneck. Although the design of such system is usually based on the moving computation to data concept to save network traffic, the contention over network resources is inevitable. In particular, we are focusing on the popular Hadoop cluster and its limitations in handling remote data access between the application and the underlying distributed file system. Two solutions are created to leverage the parallelism inside the source and destination of data traffic: a multi-sourced streaming technique that enables the application to simultaneously read input data the from multiple sources and adaptively avoid the congested network routes; and a multicast-based replication technique that reduces the network traffic in writing output data to the file system. Our experiment shows that these techniques can effectively improve the throughput of Hadoop cluster.

REFERENCES

[1] Abad, C. L., Lu, Y., and Campbell, R. H., "Dare: Adaptive data replication for efficient cluster scheduling," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pp. 159–168, Ieee, 2011.

[2] Abd-El-Malek, M., Courtright II, W. V., Cranor, C., Ganger, G. R., Hendricks, J., Klosterman, A. J., Mesnier, M. P., Prasad, M., Salmon, B., Sambasivan, R. R., and others, "Ursa minor: Versatile cluster-based storage.," in *FAST*, vol. 5, p. 163, 2005.

[3] Adamson, B., Bormann, C., Handley, M., and Macker, J., "Negative-acknowledgment (nack)-oriented reliable multicast (norm) protocol," *Internet Society Request for Comments RFC*, vol. 3940, 2004.

[4] Agarwal, V. and Rejaie, R., "Adaptive multi-source streaming in heterogeneous peer-to-peer networks," in *Electronic Imaging 2005*, pp. 13–25, International Society for Optics and Photonics, 2005.

[5] Ajima, Y., Inoue, T., Hiramoto, S., and Shimizu, T., "Tofu: Interconnect for the k computer," *Fujitsu Sci. Tech. J*, vol. 48, no. 3, pp. 280–285, 2012.

[6] Al-Fares, M., Loukissas, A., and Vahdat, A., "A scalable, commodity data center network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, 2008.

[7] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A., "Hedera: Dynamic flow scheduling for data center networks.," in *NSDI*, vol. 10, pp. 19–19, 2010.

[8] Amazon, E., "Amazon elastic compute cloud (amazon ec2)," *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.

[9] Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., and Harris, E., "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proceedings of the sixth conference on Computer systems*, pp. 287–300, ACM, 2011.

[10] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I., "Pacman: coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 20–20, USENIX Association, 2012.

[11] Andersen, R. and Vinter, B., "Harvesting idle windows cpu cycles for grid computing.," in *GCA*, pp. 121–126, Citeseer, 2006.

[12] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D., "Seti@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[13] Association, I. T. and others, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[14] Avery, C., "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[15] Ayguadé, E., Badia, R. M., Igual, F. D., Labarta, J., Mayo, R., and Quintana-Ortí, E. S., "An extension of the starss programming model for platforms with multiple gpus," in *Euro-Par 2009 Parallel Processing*, pp. 851–862, Springer, 2009.

[16] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., and others, "The nas parallel benchmarks summary and preliminary results," in *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pp. 158–165, IEEE, 1991.

[17] Ban, B., "The jgroups project."

[18] Barak, A., Ben-Nun, T., Levy, E., and Shiloh, A., "A package for opencl based heterogeneous computing on clusters with many gpu devices," in *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pp. 1 –7, sept. 2010.

[19] Barroso, L. A., Dean, J., and Holzle, U., "Web search for a planet: The google cluster architecture," *Micro, Ieee*, vol. 23, no. 2, pp. 22–28, 2003.

[20] Beaver, D., Kumar, S., Li, H. C., Sobel, J., Vajgel, P., and others, "Finding a needle in haystack: Facebook's photo storage.," in *OSDI*, vol. 10, pp. 1–8, 2010.

[21] Becker, D. J., Sterling, T., Savarese, D., Dorband, J. E., Ranawak, U. A., and Packer, C. V., "Beowulf: A parallel workstation for scientific computation," in *Proceedings, International Conference on Parallel Processing*, vol. 95, 1995.

[22] Braam, P. J. and others, "The lustre storage architecture," 2004.

[23] Buyya, R., "High performance cluster computing," *New Jersey: F'rentice*, 1999.

[24] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I., "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, vol. 25, no. 6, pp. 599–616, 2009.

[25] Chang, C., Wawrzynek, J., and Brodersen, R. W., "Bee2: A high-end reconfigurable computing system," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.

[26] Chen, Y., "Statistical workload injector for mapreduce."

[27] Chen, Y., Alspaugh, S., and Katz, R., "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[28] Cheng, Z., Luan, Z., Meng, Y., Xu, Y., Qian, D., Roy, A., Zhang, N., and Guan, G., "Erms: An elastic replication management system for hdfs," in *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*, pp. 32–40, IEEE, 2012.

[29] Chodorow, K., *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013.

[30] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I., "Managing data transfers in computer clusters with orchestra," in *SIGCOMM '11*, (New York, NY, USA), pp. 98–109, ACM, 2011.

[31] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A., "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[32] Danalis, A., Marin, G., McCurdy, C., Meredith, J., Roth, P., Spafford, K., Tipparaju, V., and Vetter, J., "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 63–74, ACM, 2010.

[33] Datasheet, V., "Vmware vcenter serverunify and simplify virtualization management," 2011.

[34] Dean, J. and Ghemawat, S., "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[35] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, ACM, 2007.

[36] DeWitt, D. and Gray, J., "Parallel database systems: the future of high performance database systems," *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[37] Diamos, G. and Yalamanchili, S., "Speculative execution on multi-gpu systems," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, IEEE, 2010.

[38] Dowty, M. and Sugerman, J., "Gpu virtualization on vmware's hosted i/o architecture," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 73–82, 2009.

[39] Duato, J., Pena, A., Silla, F., Mayo, R., and Quintana-Ortí, E., "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pp. 224–231, IEEE, 2010.

[40] Eswaradass, A., Sun, X.-H., and Wu, M., "A neural network based predictive mechanism for available bandwidth," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pp. 33a–33a, IEEE, 2005.

[41] Fan, B., Tantisiriroj, W., Xiao, L., and Gibson, G., "Diskreduce: Raid for data-intensive scalable computing," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pp. 6–10, ACM, 2009.

[42] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S., "Gpu cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 47, IEEE Computer Society, 2004.

[43] Fatica, M., "Accelerating linpack with cuda on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 46–51, ACM, 2009.

[44] Feng, W.-c., "The green500 list."

[45] Floyd, S., Jacobson, V., Liu, C.-G., McCanne, S., and Zhang, L., "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions on Networking (TON)*, vol. 5, no. 6, pp. 784–803, 1997.

[46] Gemmell, J., Montgomery, T., Speakman, T., and Crowcroft, J., "The pgm reliable multicast protocol," *Network, IEEE*, vol. 17, no. 1, pp. 16–22, 2003.

[47] George, L., *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[48] Ghemawat, S., Gobioff, H., and Leung, S.-T., "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 29–43, ACM, 2003.

[49] Giunta, G., Montella, R., Agrillo, G., and Coviello, G., "A gpgpu transparent virtualization component for high performance computing clouds," *Euro-Par 2010-Parallel Processing*, pp. 379–391, 2010.

[50] Gregg, C., Dorn, J., Hazelwood, K., and Skadron, K., "Fine-grained resource sharing for concurrent gpgpu kernels," in *4th USENIX Workshop on Hot Topics in Parallelism (HotPar). Berkeley, CA*, 2012.

[51] Grönvall, B., Westerlund, A., and Pink, S., "The design of a multicast-based distributed file system," in *OSDI*, pp. 251–264, 1999.

[52] Gropp, W., Lusk, E., and Skjellum, A., *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[53] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., and Ranganathan, P., "Gvim: Gpu-accelerated virtual machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pp. 17–24, ACM, 2009.

[54] Hadoop, A., "Apache hadoop," 2011.

[55] Hamada, T. and Nitadori, K., "190 tflops astrophysical n-body simulation on a cluster of gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–9, IEEE Computer Society, 2010.

[56] Hammoud, M., Rehman, M. S., and Sakr, M. F., "Center-of-gravity reduce task scheduling to lower mapreduce network traffic," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 49–58, IEEE, 2012.

[57] Hammoud, M. and Sakr, M. F., "Locality-aware reduce task scheduling for mapreduce," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 570–576, IEEE, 2011.

[58] Hammoud, S., Li, M., Liu, Y., Alham, N. K., and Liu, Z., "Mrsim: A discrete event based mapreduce simulator," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on*, vol. 6, pp. 2993–2997, IEEE, 2010.

[59] Henderson, R. L., "Job scheduling under the portable batch system," in *Job scheduling strategies for parallel processing*, pp. 279–294, Springer, 1995.

[60] Hetherington, T. H., Rogers, T. G., Hsu, L., O'Connor, M., and Aamodt, T. M., "Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pp. 88–98, IEEE, 2012.

[61] Hong, C.-Y., Caesar, M., and Godfrey, P. B., "Finishing flows quickly with preemptive scheduling," *SIGCOMM Comput. Commun. Rev.*, vol. 42, pp. 127–138, Aug. 2012.

[62] Hong, C., Chen, D., Chen, W., Zheng, W., and Lin, H., "Mapcg: writing parallel program portable between cpu and gpu," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 217–226, ACM, 2010.

[63] Huang, J., Ouyang, X., Jose, J., Wasi-ur Rahman, M., Wang, H., Luo, M., Subramoni, H., Murthy, C., and Panda, D. K., "High-performance design of hbase with rdma over infiniband," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 774–785, IEEE, 2012.

[64] Iancu, C., Hofmeyr, S., Blagojevic, F., and Zheng, Y., "Oversubscription on multicore processors," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –11, april 2010.

[65] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A., "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 261–276, ACM, 2009.

[66] Islam, N. S., Rahman, M., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., Murthy, C., and Panda, D. K., "High performance rdma-based design of hdfs over infiniband," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 35, IEEE Computer Society Press, 2012.

[67] Jackson, D., Snell, Q., and Clement, M., "Core algorithms of the maui scheduler," in *Job Scheduling Strategies for Parallel Processing*, pp. 87–102, Springer, 2001.

[68] Jeffers, J. and Reinders, J., *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[69] Jiménez, V. J., Vilanova, L., Gelado, I., Gil, M., Fursin, G., and Navarro, N., "Predictive runtime code scheduling for heterogeneous architectures," in *High Performance Embedded Architectures and Compilers*, pp. 19–33, Springer, 2009.

[70] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P., Madden, S., Stonebraker, M., Zhang, Y., and others, "H-store: a high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[71] Kayyali, B., Knott, D., and Van Kuiken, S., "The big-data revolution in us health care: Accelerating value and innovation," *Mc Kinsey & Company*, 2013.

[72] Kim, J., Kim, H., Lee, J., and Lee, J., "Achieving a single compute device image in opencl for multiple gpus," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pp. 277–288, ACM, 2011.

[73] Kim, J., Dally, W. J., Scott, S., and Abts, D., "Technology-driven, highly-scalable dragonfly topology," in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 77–88, IEEE Computer Society, 2008.

[74] Kolberg, W., Marcos, P. D. B., Anjos, J. C., Miyazaki, A. K., Geyer, C. R., and Arantes, L. B., "Mrsg–a mapreduce simulator over simgrid," *Parallel Computing*, vol. 39, no. 4, pp. 233–244, 2013.

[75] Kreps, J., Narkhede, N., Rao, J., and others, "Kafka: A distributed messaging system for log processing," in *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[76] Kronenberg, N. P., Levy, H. M., and Strecker, W. D., "Vaxcluster: a closely-coupled distributed system," *ACM Transactions on Computer Systems (TOCS)*, vol. 4, no. 2, pp. 130–146, 1986.

[77] Kwon, J. B. and Yeom, H. Y., "Distributed multimedia streaming over peer-to-peer networks," in *Euro-Par 2003 parallel processing*, pp. 851–858, Springer, 2003.

[78] Lakshman, A. and Malik, P., "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[79] Lee, E. K. and Thekkath, C. A., "Petal: Distributed virtual disks," in *ACM SIGOPS Operating Systems Review*, vol. 30, pp. 84–92, ACM, 1996.

[80] Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y. D., and Moon, B., "Parallel data processing with mapreduce: a survey," *AcM sIGMoD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[81] Li, T., Narayana, V. K., El-Araby, E., and El-Ghazawi, T., "Gpu resource sharing and virtualization on high performance computing systems," in *Parallel Processing (ICPP), 2011 International Conference on*, pp. 733–742, IEEE, 2011.

[82] Li, T., Narayana, V. K., and El-Ghazawi, T., "A static task scheduling framework for independent tasks accelerated using a shared graphics processing unit," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pp. 88–95, IEEE, 2011.

[83] Lin, J. C. and Paul, S., "Rmtp: A reliable multicast transport protocol," in *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, vol. 3, pp. 1414–1424, IEEE, 1996.

[84] Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H., "Merge: a programming model for heterogeneous multi-core systems," in *ACM SIGOPS operating systems review*, vol. 42, pp. 287–296, ACM, 2008.

[85] Liu, Y., Li, M., Alham, N. K., and Hammoud, S., "Hsim: a mapreduce simulator in enabling cloud computing," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 300–308, 2013.

[86] Lohr, S., "The age of big data," *New York Times*, vol. 11, 2012.

[87] Luk, C.-K., Hong, S., and Kim, H., "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 45–55, IEEE, 2009.

[88] Macker, J. P. and Adamson, R. B., "A tcp friendly, rate-based mechanism for nack-oriented reliable multicast congestion control," in *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE*, vol. 3, pp. 1620–1625, IEEE, 2001.

[89] Merritt, A., Gupta, V., Verma, A., Gavrilovska, A., and Schwan, K., "Shadowfax: scaling in heterogeneous cluster systems via gpgpu assemblies," in *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, pp. 3–10, ACM, 2011.

[90] Minelli, M., Chambers, M., and Dhiraj, A., *Big data, big analytics: emerging business intelligence and analytic trends for today's businesses*. John Wiley & Sons, 2012.

[91] Nadkarni, A. and DuBois, L., "Trends in enterprise hadoop deployments," in *Survey*, IDC, 2013.

[92] Nguyen, T. and Zakhor, A., "Distributed video streaming over internet," in *Proc. of Multimedia Computing and Networking (MMCN02)*, 2002.

[93] Nickolls, J. and Dally, W., "The gpu computing era," *Micro, IEEE*, vol. 30, pp. 56 –69, march-april 2010.

[94] Nickolls, J., Buck, I., Garland, M., and Skadron, K., "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[95] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A., "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1099–1110, ACM, 2008.

[96] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., and others, "The case for ramclouds: scalable high-performance storage entirely in dram," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.

[97] Owen, S., Anil, R., Dunning, T., and Friedman, E., *Mahout in action*. Manning, 2011.

[98] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., and Phillips, J., "Gpu computing," *Proceedings of the IEEE*, vol. 96, pp. 879 –899, may 2008.

[99] Palanisamy, B., Singh, A., Liu, L., and Jain, B., "Purlieus: locality-aware resource allocation for mapreduce in a cloud," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 58, ACM, 2011.

[100] Panetta, J., Teixeira, T., de Souza Filho, P. R., da Cunha Finho, C. A., Sotelo, D., Da Motta, F., Pinheiro, S. S., Pedrosa, I., Rosa, A. L. R., Monnerat, L. R., and others, "Accelerating kirchhoff migration by cpu and gpu cooperation," in *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*, pp. 26–32, IEEE, 2009.

[101] Pfister, G. F., *In search of clusters*. Prentice-Hall, Inc., 1998.

[102] Phull, R., Li, C.-H., Rao, K., Cadambi, H., and Chakradhar, S., "Interference-driven resource management for gpu-based heterogeneous clusters," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pp. 109–120, ACM, 2012.

[103] Pienaar, J. A., Raghunathan, A., and Chakradhar, S., "Mdr: performance model driven runtime for heterogeneous parallel platforms," in *Proceedings of the international conference on Supercomputing*, pp. 225–234, ACM, 2011.

[104] Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., and Stoica, I., "Faircloud: sharing the network in cloud computing," in *SIGCOMM '12*, (New York, NY, USA), pp. 187–198, ACM, 2012.

[105] Qiao, Y., Skicewicz, J., and Dinda, P., "An empirical study of the multiscale predictability of network traffic," in *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pp. 66–76, IEEE, 2004.

[106] Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and Handley, M., "Improving datacenter performance and robustness with

multipath tcp," in *SIGCOMM '11*, (New York, NY, USA), pp. 266–277, ACM, 2011.

[107] Ravi, V. T., Becchi, M., Agrawal, G., and Chakradhar, S., "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th international symposium on High performance distributed computing*, pp. 217–228, ACM, 2011.

[108] Rizzo, L., "pgmcc: a tcp-friendly single-rate multicast congestion control scheme," in *ACM SIGCOMM Computer Communication Review*, vol. 30, pp. 17–28, ACM, 2000.

[109] Ronstrom, M. and Thalmann, L., "Mysql cluster architecture overview," *MySQL Technical White Paper*, 2004.

[110] Sajjapongse, K., Wang, X., and Becchi, M., "A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pp. 179–190, ACM, 2013.

[111] Sengupta, D., Belapure, R., and Schwan, K., "Multi-tenancy on gpgpu-based servers," in *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, pp. 3–10, ACM, 2013.

[112] Seo, S., Jang, I., Woo, K., Kim, I., Kim, J.-S., and Maeng, S., "Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1–8, IEEE, 2009.

[113] Shvachko, K., Kuang, H., Radia, S., and Chansler, R., "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[114] Smith, G., "Oracle rac 10g overview," *An Oracle White Paper, Oracle Corporation*, pp. 1–15, 2003.

[115] Spafford, K., Meredith, J., and Vetter, J., "Maestro: data orchestration and tuning for opencl devices," in *Euro-Par 2010-Parallel Processing*, pp. 275–286, Springer, 2010.

[116] Stamatakis, A., "Raxml-vi-hpc: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinformatics*, vol. 22, no. 21, pp. 2688–2690, 2006.

[117] Staples, G., "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 8, ACM, 2006.

[118] Stone, J. E., Gohara, D., and Shi, G., "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[119] Strickland, J. W., Freeh, V. W., Ma, X., and Vazhkudai, S. S., "Governor: Autonomic throttling for aggressive idle resource scavenging," in *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pp. 64–75, IEEE, 2005.

[120] Strohmaier, E., "The top500 supercomputer sites."

[121] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., and Shah, S., "Serving large-scale batch computed data with project voldemort," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 18–18, USENIX Association, 2012.

[122] Teodoro, G., Sachetto, R., Sertel, O., Gurcan, M. N., Meira, W., Catalyurek, U., and Ferreira, R., "Coordinating the use of gpu and cpu for improving performance of compute intensive applications," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1–10, IEEE, 2009.

[123] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., and Murthy, R., "Hive-a petabyte scale data warehouse using hadoop," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 996–1005, IEEE, 2010.

[124] Venkatasubramanian, S., Vuduc, R. W., and others, "Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 244–255, ACM, 2009.

[125] Vetter, J. S., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., and others, "Keeneland: Bringing heterogeneous gpu computing to the computational science community," *Computing in Science and Engineering*, vol. 13, no. 5, pp. 90–95, 2011.

[126] Volkov, V. and Demmel, J., "Lu, qr and cholesky factorizations using vector capabilities of gpus," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May*, pp. 2008–49, 2008.

[127] Wang, G., Butt, A. R., Pandey, P., and Gupta, K., "A simulation approach to evaluating design decisions in mapreduce setups," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pp. 1–11, IEEE, 2009.

[128] Wang, R. Y. and Anderson, T. E., "xfs: A wide area mass storage file system," in *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pp. 71–78, IEEE, 1993.

[129] Wang, Y., Que, X., Yu, W., Goldenberg, D., and Sehgal, D., "Hadoop acceleration through network levitated merge," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 57, ACM, 2011.

[130] Wei, Q., Veeravalli, B., Gong, B., Zeng, L., and Feng, D., "Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pp. 188–196, IEEE, 2010.

[131] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C., "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320, USENIX Association, 2006.

[132] Widmer, J. and Handley, M., "Tcp-friendly multicast congestion control (tfmcc): Protocol specification," tech. rep., RFC 4654, August, 2006.

[133] Wu, H., Diamos, G., Cadambi, S., and Yalamanchili, S., "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 107–118, IEEE Computer Society, 2012.

[134] Wu, J. and Hong, B., "Collocating cpu-only jobs with gpu-assisted jobs on gpu-assisted hpc," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pp. 418–425, IEEE, 2013.

[135] Wu, J. and Hong, B., "Improving mapreduce performance by streaming input data from multiple replicas," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 1, pp. 623–630, IEEE, 2013.

[136] Wu, J. and Hong, B., "Multi-source streaming-based data accesses for mapreduce systems," *International Journal of Big Data Intelligence*, vol. 1, no. 1, pp. 36–49, 2014.

[137] Wu, J. and Hong, B., "Multicast-based replication for hadoop hdfs," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, pp. 1–6, IEEE, 2015.

[138] Wu, J., Shi, W., and Hong, B., "Dynamic kernel/device mapping strategies for gpu-assisted hpc systems," in *Job Scheduling Strategies for Parallel Processing*, pp. 96–113, Springer, 2013.

[139] Xu, D., Hefeeda, M., Hambrusch, S., and Bhargava, B., "On peer-to-peer media streaming," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pp. 363–371, IEEE, 2002.

[140] Yuan, Y., Lee, R., and Zhang, X., "The yin and yang of processing data warehousing queries on gpu devices," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 817–828, 2013.

[141] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I., "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, pp. 265–278, ACM, 2010.

[142] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I., "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10–10, 2010.

[143] Zawodny, J., "Redis: Lightweight key/value store that goes the extra mile," *Linux Magazine*, vol. 79, 2009.

[144] Zhong, J. and He, B., "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1522–1532, 2014.